A Framework for Network Intrusion Detection on Open Platform Communications Unified Architecture

Masterarbeit
Master-Thesis von Tomas Bortoli aus Gavardo
Tag der Einreichung:

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Frank Weber
3. Gutachten: Ing. Pedro Larbig

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Sicherheit Informatik
Fraunhofer SIT

A Framework for Network Intrusion Detection on Open Platform Communications Unified Architecture
Masterarbeit

Vorgelegte Master-Thesis von Tomas Bortoli aus Gavardo

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Frank Weber
3. Gutachten: Ing. Pedro Larbig

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 19th September 2017

(T. Bortoli)

Contents

List of Figures

Abstract

Open Platform Communications Unified Architecture (OPC UA) is a Machine to Machine (M2M) communication standard, first released in 2008 as the evolution of OPC, created for Industrial Control Systems (ICS) and Internet of Things (IoT) programming. It was designed to create an abstract model on which any information exchange in form of structured data can be implemented. Industry and state actors use it to control factories and plants thus putting OPC UA dependent software in a critical security position. In December 2015, the German Federal Office for Information Security proved that an official reference implementation of OPC UA contained security flaws in the code that could compromise, if exploited, industrial machineries and other dependent systems [49]. Cyber attacks in ICS may be extremely expensive because of the critical processes which they aim to stop.

This thesis proposes a Network Intrusion Detection System (NIDS) based solution to monitor malicious computer attacks on OPC UA. This work develops a plug-in for the dynamic Bro NIDS to support OPC UA based protocols, therefore it creates an Application Programming Interface (API) that can be used to write Turing complete security policies in the Bro language. Furthermore, policy scripts have been implemented to detect the exploitation of flaws and standard inconsistencies found in the analysis [49]. In addition, the parser is also able to detect malformed packets, also sources of attacks in general and those identified in [49]. The result has been tested and evaluated in efficiency, security and standard coverage terms. The aim of this project is to suggest the use of an additional tool that might be used by Computer Emergency Response Teams (CERTs) to investigate any attack and in order to safeguard OPC UA dependent machines.

Preface

This master thesis tackles the problem of having a Network Intrusion Detection System (NIDS) that supports the OPC UA open standard (IEC 62541). Writing a NIDS from scratch would require too much time and therefore this work exploits the powerful and extensible Bro NIDS. This work develops a plug-in for Bro, written in Binpac, to unleash Bro's potent policy script engine. Therefore allowing to actually develop Turing complete security policy scripts in the Bro language to detect malicious actions in the traffic.

The introduction of this master thesis informs and discusses Industrial Control Systems (ICS), NIDS and OPC UA. These topics are given respectively in sections 1.1, 1.3, 1.2. State of the art of the IEC 62541 open standard is discussed in section 2.1.

An introduction to the potentiality of Bro NIDS is given in section 2.2. Furthermore, in part 2.3 is discussed Binpac, a special compiler that allows to write application layer protocol parsers in the Domain Specific Language (DSL) also named Binpac, to integrate OPC UA parsing in Bro. In the appendix D are listed the API that the plug-in provides. Data structures used in the API instead, are listed in appendix C.

Design and implementation of the plug-in and policy scripts are in sections 3.1.1 and 3.2.1, respectively. Moreover, the work develops few detection scripts to show the usefullness of the plug-in to identify attacks and deviations from the standard in real-time (starting from section 3.2.4 to 3.2.7). The implemented parser has also the capability of detecting malformed packets that are normally used during the run of fuzz testing or in outbreaks (section 3.2.8). Finally, evaluation and conclusions of the work are provided respectively in sections 3.3 and 4.

1 Introduction

This is the introduction section, in which disparate concepts are presented to the reader with the purpose of informing him on topics and terminology that are discussed in this thesis.

1.1 Industrial Control System (ICS)

Industrial Control System (ICS) is a generic term that fundamentally identifies all the control systems used in industrial production technologies. ICS was born with the digitalization of the industries but nowadays it includes a huge variety of technologies such as: Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control Systems (DCS) and other systems based on Programmable Logic Controllers (PLC) [15].

ICS are typically employed in industry sectors such as electrical, water, oil, natural gas, chemical, transportation, food, beverage, automotive, aerospace and others, although not all the applications are in the industry sector [15]. These systems are usually critical to the normal functioning of industries and plants. Therefore, several components of the modern society actually relay on them. Critical infrastructures control the distribution of electric energy among the other things. If for some reason the current stops flowing in the region of a country, almost all the electric dependent activities would immediately be compromised.

ICS technologies are extremely spread and fragmented because they have to cope with all the necessities of the wide modern industry. There are hundreds of ICS protocols (i.e.: Profinet, Profibus, Modbus), classified by families and goals. Most of them are proprietary protocols, that are not released to the public but to industry partners. Another example is the S7comm, that is a proprietary protocol from Siemens that runs between PLCs systems [35]. Other protocols might require the involved parties to buy the specifications, i.e., the Fieldbus family of protocols [19]. Furthermore, ICS systems usually require a high efficiency, having real-time constraints [15]. Because of the specificity of ICS software, hardware, its costs and constraints, it follows that ICSs are a huge branch of technology that is mostly handled by both, the private sector and state actors [15].

Security in ICS

Security in ICS is a fundamental issue and basic guidelines have been defined in [15] by the National Institute of Standard and Technology (NIST) based also on [43]. There is also an Industrial Control System Cyber Emergency Response Team (ICS-CERT) [20] managed by the department of homeland security of the United States government. Their work aims at reducing the risks related to all critical infrastructure sectors by cooperating with federal and intelligence agencies, control system owners, operators and vendors. They provide alerts for those concerned with the current threats to critical infrastructure networks and security advisories about the new vulnerabilities discovered in such systems. As of May 2017, more than 50 security advisories have been published on the ICS-CERT website for the year 2017 [20]. This shows that there are security researchers, penetration testers and auditors that are actively seeking for new flaws. The ICS-CERT also provides other services: training to prepare professionals to the world of ICS.

ICS security is a delicate topic because of all the industry and critical infrastructure that depends on it and because initially, ICS were different from information technology (IT) systems because of the typical isolation between them and the rest of the IT network [15]. Nowadays instead, they are becoming more integrated into IT networks [15] for several reasons. This change exposes them to more threats coming from both, the Internet and local networks.

Connecting computer devices to the network bring several benefits, i.e.: advantage of remote control, data collection, real-time monitoring and other useful features but is also risky in security terms. Issues arise especially in embedded devices because they usually run a custom firmware written in some low level programming language, likely C, tremendously prone to errors. Moreover, embedded devices do not usually support the latest security features that are instead supported on chipsets placed in conventional business laptops, desktop and server computers. The latest chipset's security feature that might be missing are: Supervisor Mode Access Protection (SMAP), Supervisor Mode Execution Prevention (SMEP), Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) [50]. These features require tight cooperation between the operating system and the hardware chipset. The lack of at least one of the cited features represents the lack of a modern hardware/software adequate security protection mechanisms. Moreover, embedded devices might not have a separation between user space and kernel space, running all the code with the same ring privileges, therefore making SMAP and SMEP protections impossible to be implemented. This thesis will discuss more about specific problems of embedded devices in section 2.1.2.

ICS devices are delicate from a security perspective. Denial of Service (DoS) attacks are usually easier to achieve than outbreaks that get hold of the system. However, DoS attacks can already put at risk the manufacturing process if they hit the production plant [62]. Therefore, ICS devices, parts of the critical infrastructure [15], have to be carefully designed, developed and tested by companies and state actors. In a cyber war scenario it would be strategic to tear down the enemies' ICS production or the critical infrastructures' stability by exploiting flaws in software implementations of OPC UA or other ICS protocols.

1.2 Open Platform Communications (OPC)

Open Platform Communications (OPC), or Object Linking and Embedding (OLE) for Process Control (OPC) is a series of standards specifications, developed with the collaboration of leading worldwide automation producers, working in cooperation with Microsoft [26]. The name is taken by the fact that originally it was based on Microsoft's OLE Component Object Model (COM) and Distributed Component Object Model (DCOM) [26]. The specifications defined a set of objects, interfaces and methods to be used in product manufacturing and industry automation.

Several protocols have been developed as part of the OPC serie. DCOM based protocols standardized by OLE are: the OPC Data Access protocol (OPC DA), the OPC Historical Data Access protocol (OPC HDA), OPC Alarm & Events protocol (OPC AE) and OPC Commands protocols [26]. There is also a Simple Object Access Protocol (SOAP) based protocol, OPC XML-DA, the first web service based OPC protocol [26].

Despite its advantages, the development of OPC is terminated, mainly because of its dependency to Microsoft based technology. But there were also other bad aspects in OPC itself, as bad security and limitations on the information model descriptive capacity. It does not provide the flexibility that it is needed in the current technologies [27] [30]. At its place OPC Unified Architecture has been introduced.

OPC Unified Architecture (OPC UA)

OLE for Process Control Unified Architecture (OPC UA), released in 2008 [33], is not just a protocol but a reinvention of OPC itself. The OPC foundation defined a new information model which allows arbitrary data to be conveyed with it. One of the most important features is that it is completely portable and therefore not bound anymore to Microsoft [27]. This new set of standards is referred with the IEC 62541 abbreviation.

OPC UA defines standards to allow the communication on devices (between different components of a machinery itself), between machines and from machines to other IT systems, connecting IT and Operational Technology (OT) systems. In addition, it is platform independent, i.e, OPC UA can be used from Windows, GNU/Linux, OSX and Android systems. It can run on embedded devices, as well as on mobile devices and also on the cloud. OPC UA is supposed to be used on local networks as well as on the Internet; therefore it provides secure tunnels for the connections [33].

Implementation of the standard at issue has already been widely integrated into industries like: automotive, beverages, packaging, oil, automation and others because of its flexibility and extensibility [33].

OPC UA describes services on an abstract level, which can then be associated to different protocols. One of the most important is OPC UA-TCP or "OPC Binary". It is basically a binary encoding of the data transported upon a standard Transmission Control Protocol (TCP)/Internet Protocol (IP) stack. OPC UA-TCP is mapped to port 4840 by the Internet Assigned Numbers Authority (IANA) [18]. Then, there is a secure version of OPC UA-TCP that was called OPC UA-Transport Layer Security (TLS) that is officially running on port 4843 [18].

The adoption by many industry companies of OPC UA is moving the ICS sector towards producing more Free and Open Source Software (FOSS) [29][28][23].

Another important characteristic of OPC UA is that it is defined so generically and with enough flexibility so that it is also supposed to be used with the Internet of Things (IoT) devices [32] [57]. Therefore, it gives interoperability between the industry and the IoT, hence interconnecting these two different technologies.

Figure 1 shows an example of possible OPC UA architecture:

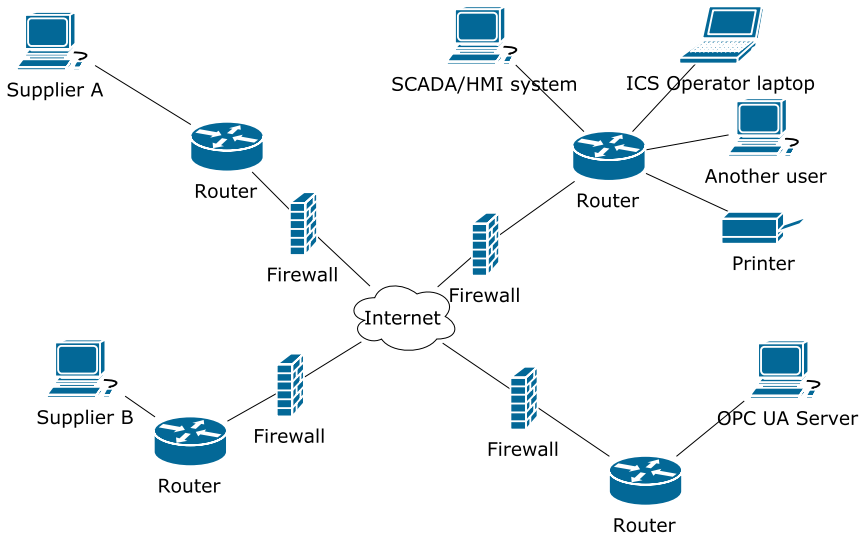

Figure 1: OPC UA example architecture

The picture depicts an hypothetical usage scenario for OPC UA in which there are servers and clients distributed across the Internet. IoT devices on the network can make use of OPC UA to share data between them and with other systems thanks to the OPC UA's implementations.

1.3 Network Intrusion Detection System

A Network Intrusion Detection System (NIDS) is a computer system, usually implemented in software, whose main role is to detect attacks. Attack means every attempt to unauthorizedly undermine the stability of the Confidentiality, Integrity and Availability (CIA) of the assets on the network [52]. Further goals of such a system are to react to the intrusions, for example by notifying network administrators or others that such intrusions are happening and to record the associated packets for further inspection [52]. Potentially a NIDS could also prevent the outbreak by dropping the malicious packets before they reach the supposed destination. This can have side effects if the dropped packets are not malicious.

In a real-world scenario, usually, network intrusion detection systems are employed not as active packets filters but as systems of analysis and research of new attack techniques [65]. These systems can be used to implement advanced monitoring of the network traffic [53]. Ethical preserving purposes that might be achieved using modern NIDS are: detecting intrusions and suspicious traffic for further inspection. Suspicious traffic means packets that violates some defined security policies and/or packets whose structure differs from the expected one (malformed packet). Once new attacks are detected, the exploited flaws can be studied by reproducing the outbreak while debugging the program to understand what is/are the exploited bug(s). After the fixes are applied and deployed, a security advisory might be published to inform customers, security researchers, etc, that a flaw was found and fixed.

The Bro NIDS can also be used to accomplish tasks such as several types of advanced network surveillance [61]. Network monitoring software can be easily used to oversee on unencrypted network traffic. Nonetheless, most of the data users send through the web, especially sensible data, are encrypted and secured by the Hyper Text Transfer Protocol Secure (HTTPS) that uses trusted Certificate Authorities (CA) to build chains of trust to the sites' certificates. However, some messages, for example the Domain Name System (DNS) queries are still conveyed in clear text and that basically informs network monitors of which servers are being contacted by whom in the network. However, even without DNS, network monitors would still be able to know which servers the hosts are connecting to by making reverse DNS queries with the server's IP, that is encoded in plaintext in the headers of the IP packets.

NIDS usually run on some critical nodes on a network where all the traffic passes through, or they operate with a copy of the packets in transit to avoid the risk to slowdown the network operations. However, a NIDS required to inspect all the traffic of a network might not meet the performance requirements and it could get stuck because of resource exhaustion. That is why it is important to have some experts that monitor and read the log of NIDS after successful installation.

The most advanced existing NIDS, up to the author's knowledge are Bro[61] and Snort[63]. The last is a lightweight, simple NIDS that does not really fit the scenario of this thesis because it misses the capability of Bro of stateful processing

of the parsed information. Furthermore, Snort lacks a handly mechanism to extend its parser. If a new protocol needs to be supported then the source code of Snort has to be adapted.

In the next two sections are presented brief descriptions of the core features and characteristics of the two presented network intrusion detection systems, Bro and Snort, respectively in sections 1.3.2 and 1.3.1.

1.3.1 Snort

Snort is a signature based network intrusion detection system. These kind of NIDS are based on the concept of recognizing particular packets through the so called "signatures" and then raise a warning if the packet matches. Therefore, Snort allows to enforce security policies through static expressions applied on the content of the packets. Furthermore, it is possible to program Snort using Berkley Software Distribution (BSD) packets filters to filter the traffic. Then, Snort applies expressions to decide whether a packet has to raise an alert or not [63].

The idea behind signature based network intrusion detection systems is that using a significant description of a particular cyber attack, it becomes possible to detect future outbreaks of that kind by checking for its signature (static expressions) [63]. An evasion mechanism for signature based NIDS is i.e. when an attacker mutates the payload, the padding and other components of his exploit to bypass the detection mechanism. Moreover, in a typical scenario, the "signatures" are publicly available, as well as the NIDS source code. Therefore, an attacker can practice an attack until its success rate is satisfactory, before using it against the target.

Snort however has been surpassed by Bro in terms of flexibility because the last supports more protocols, a plug-in system for expanding its parsing capabilities and a security policy scripting engine to give statuful processing of the packets' content.

1.3.2 Bro

Bro is a hybrid network intrusion detection system. Bro's project is being developed by an open communicaty. There is a public mailing list in which people can communicate, ask questions, report bugs and discuss about documentations, etc. Furthermore, there are many other projects regarding the development of additional features and tools for Bro. Last but not least, Bro is very powerful, it already supports 52 network protocols [4] and it provides worthy means of expanding and exploiting its current capabilities.

The official mailing list is the public mean of contact between the Bro's users and developers. It is also the place where people can ask questions, help, suggest improvements, report bugs, etc. The mailing list is also used to talk about the all the projects under development. Moreover, proposals for changes from anybody are welcome if targeting specific issues or with good motivation.

The main advantages of Bro over its competitor Snort, discussed in section 1.3.1, are two: first, Bro allows stateful, Turing complete processing of the flowing packets, using a domain specific language similar to "C++", called Bro, that has been designed for that purpose [61]. And second, Bro allows to write plug-ins using a compiler for application layer protocol parsers called Binpac (section 2.3). Therefore, developers can use Binpac to implement parsers for new protocols and create high-level interfaces to allow policy scripts to process new protocols [64].

Bro is a hybrid system, signature and anomaly based. It abstracts the problem of parsing from the security policy processing. The two operations are performed in two different steps using disparate instruments. The parser can be implemented in Binpac, a language and compiler that has been created for that purpose. Binpac is further discussed in section 2.3.

Bro might run on a cluster of servers that a company may dispose for it. In fact there is a project in development, branched from Bro, called "Deep Cluster" that aims at building and controlling hierarchies of clusters with the purpose of detecting intrusions, using the tool called "BroControl" [6]. This is not the classical scenario, but it has to be considered because running Bro on powerful servers is probably the only way to process really big amounts of data, that might be the output of networks with hundreds or thousands of active machines, like universities and big companies.

1.4 Problem Statement

Industrial Control Systems (ICS) have been designed and developed since the digitalization of industries and critical infrastructures [15]. In fact, the software behind those systems is mostly proprietary because developed by companies

for other businesses or governments. Therefore, the public was never involved in the development on this critical slice of technology as happened with other types of software. But as previously discussed, with the development and integration of OPC UA, the possibilities to have a more open ICS in the future are increasing.

A possible observation of the state of the art NIDS technology for ICS is that most of the support is missing, especially for protocols like OPC UA. But NIDS are very useful; they can reveal suspicious packets in real-time and alert the responsible administrators. They are critical to detect attacks in real-time and for example inform the Computer Emergency Response Team (CERT) responsible for the targeted systems. Therefore NIDS give a chance to the CERT to prevent future outbreaks by investigating and fixing the identified issues and avert successful attacks from spreading on other systems by detaching infected machines from the network. Moreover, recording and detecting attacks is advantageous to reproduce the attack and to consequently understand which flaws the offender exploited. Once the vulnerabilities are known, they can be fixed, future outbreak prevented and relative security advisories, eventually, published. As an example of the fact that current NIDS has a very limited support for ICS protocols, it is possible to observe that the Bro NIDS only supports one ICS protocol, Modbus [3].

The present work, hence, proposes to design and implement an extension for the Bro NIDS to parse and analyze protocols created using the new, open standard named OPC UA, therefore improving the state of the art of NIDS in the contextes of ICS and IoT.

The writing of a plug-in for the NIDS Bro to support OPC UA, implies implementing a high-level API that will be directly interfaced to policy scripts. These will have the form of generated events that will be caught by Bro policy scripts. Any developer will be able to use the plug-in described in this thesis to implement custom policy scripts, following the event set provided as API and the relative data structures. Furthermore, the present thesis describes, implements and tests detection techniques for real attacks, taking the examples from the security analysis of OPC UA steered by the German government [49] to show the effectiveness of a NIDS in this context.

2 State of the Art

In this section several state of the art regarding computer technologies are discussed. The current specification of the IEC 62541 standard, that is also known as OPC UA, first released in 2008 is discussed in 2.1. OPC UA is a service oriented, platform independent architecture that integrates the old functionalities of OPC Classic's specification in a new extensible framework [33].

Bro NIDS and its policy scripting language are respectively in sections 2.2 and 2.2.3. The Binpac compiler for network application layer protocol parsers is instead discussed in setion 2.3.

2.1 OPC UA

OPC UA, has been created and managed by the OPC foundation and it is an open standard rich of features. The following lists have been taken from the official OPC foundation resources with the purpose of giving a clear and genuine overview of the capabilities [33].

General OPC UA specification lines:

- Functional equivalence: all COM OPC Classic specifications are support by OPC UA
- Platform independence: from an embedded micro-controller to cloud-based infrastructure
- Secure: encryption, authentication, and auditing
- Extensible: ability to add new features without affecting existing applications
- Comprehensive information modeling: for defining complex information (i.e arbitrary data structures)

OPC UA functionalities:

- Discovery: find the availability of OPC Servers on local PCs and/or networks
- Address space: a hierarchy of data that can be browsed and utilized by OPC Clients
- Clients can read and write data based on access-permissions
- Subscriptions: monitor data and report-by-exception when values change based on a clientâĂŹs criteria
- Events: notify periodically or on changes of data to subscribed clients
- Methods: clients can execute programs, etc. based on methods defined on the server

Platform independence:

- Hardware platforms: traditional PC hardware, cloud-based servers, PLCs, micro-controllers (ARM etc.)
- Operating Systems: Microsoft Windows, Apple OSX, Android, or any distribution of Linux, etc.

Several security functions such as modern encryption and signature algorithms are provided to protect the CIA of the relative data and systems.

- Transport: a TLS layer can be employed to provide security features
- Session Encryption: messages are transmitted securely at 128 or 256 bit encryption levels
- Message Signing: verify that the data integrity is preserved after the trasmission, detects tampering
- Sequenced Packets: exposure to message replay attacks is eliminated with sequencing
- Authentication: each UA client and server is identified through OpenSSL certificates providing control over which applications and systems are permitted to connect with each other
- User Control: applications can require users to authenticate (login credentials, certificate, etc.) and can further restrict and enhance their capabilities with access rights and address-space "views"

- Auditing: activities by user and/or system are logged providing an access audit trail

2.1.1 Overview

OPC UA Overview and Concepts is the first specification of the standard (IEC 62541) [44][45][46][47][48]. This section is based on the information in [44].

OPC UA is applicable to components in all the industry domains, such as industrial sensors and actuators, control systems, manufacturing systems, including the Industrial Internet of Things (IIoT) as well as Industrie 4.0. The standard defines a common infrastructure model that OPC UA aware parties can use to exchange information. Essential components of the system are:

- A model to represent the structures, behaviors and semantics
- A message model that the applications use to interact
- A communication model to convey information between end-points
- A conformance model to guarantee interoperability

In a typical OPC UA scenario, there could be a PLC that controls a machine by taking the input from the Human Machine Interface (HMI), while, at the same time, it communicates with a back-end server with the purpose of, i.e., logging the received commands. OPC UA facilitates the interoperability of disparate machines running different platforms.

In the OPC UA universe, various systems exchange information by sending *request* and *response* messages between *clients* and *servers* or *Network Messages* between *Publishers* and *Subscribers*.

Data are transmitted using *standard defined types* and *vendor defined types*. Furthermore, the standard defines a set of *services* that should be available in the client-server architecture. Clients can ask to servers to retrieve current and past data. Servers can include *Alarms* & *Events* in responses, to inform clients. Additionally, through a concept called, *AddressSpace*, the clients can query the servers for the metadata that describe the format of the data, therefore making OPC UA format independent.

OPC UA's servers can inform clients of new types with data types definitions dynamically. Additionally, a server can provide multiple services to clients to integrate the distribution of data, alarms & events and history (past transmitted data) from a single machine. As stated previously, the standard allows data to be conveyed in different composition, between these are included the binary, XML and JSON formats.

IEC 62541 is flexible, in fact, disparate clients can get data from a server in different formats. A feasible scenario for OPC UA is one in which there are several machines, some of which are actual ICS and requires a certain specific, maybe proprietary format that can be conveyed with OPC UA, while other machines may be normal desktop computers, that are there to control the machines and supervise the process. These machines ask for a different format, based on the support of the implementation running on it (e.g JSON, XML)[44].

OPC UA also provides mechanisms for clients to quickly recover from communication failures without having to rely on the timeouts provided by the underlying protocols. This is very important because ICS applications usually have real-time constraints in the range of milliseconds [15].

A feature of the standard that gives to it a lot of flexibility is the fact that it has been designed to support a wide range of servers. Characteristics of these servers are resources, like memory, disks space, CPU performances but also execution platforms and functional capabilities. Therefore IEC 62541-7 defines *Profiles* to which servers may claim conformance. Then, clients can discover the profiles of a server and based their interactions on the *profile* features[44].

OPC UA defines also a *Publisher Subscriber* model, in which it is possible to have the communications on a peer to peer communication paradigm. Those transmissions might use UDP instead of TCP as support protocol. Such a scheme might well suit scenarios in which there are small but frequent transmissions of data. For example, it is possible to have traffic between PLC and PLC and HMIs. In such a scenario, peers are not even connected and do not need to know about the existence of each other (using UDP). Therefore, UDP based transmissions may be point-to-point as well as multicast.

OPC UA servers can also stream data to applications residing on the cloud. That might be useful for applications like: big data analytics, system optimizations and predictive maintenance.

The IEC 62541 standard is composed by disparate, independent, layers of specifications. Therefore, the core design is independent from the computing technology and the type of network transport. In this way OPC UA is autonomous respect to the platform and the machine's architecture. Furthermore, specifications' independence from current technologies will foster their porting in the future. Also, OPC UA is constructed so that the deprecated OPC COM clients are able to upgrade to the current standard natively, or using external wrappers. In OPC, each OPC COM application has its own *address space* model and set of *services*; OPC UA unifies the previous model by integrating the existing models therefore giving backward compatibility and encouraging users to facilitate the upgrade of their systems.

The standard is defined so that it is possible to have more clients concurrently connected to more servers. Figure 2 depicts a scenario in which more clients are connected to more servers.

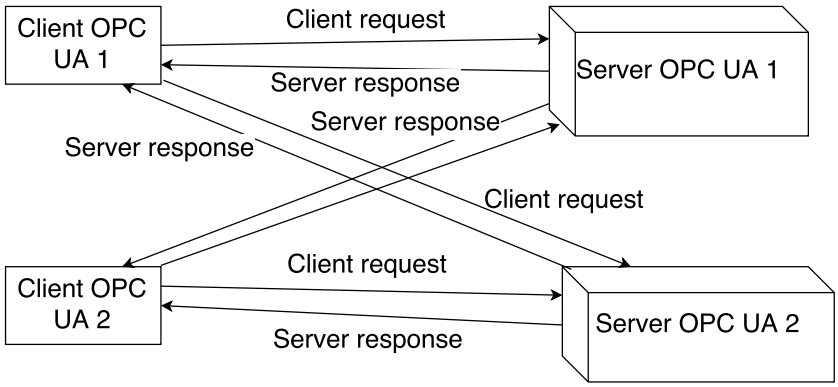

Figure 2: OPC UA Client server architecture

OPC UA clients use client Application Program Interface (API) to send and receive OPC UA service requests and responses to the OPC UA server.

Figure 3 describes the main internal components of an OPC UA server.

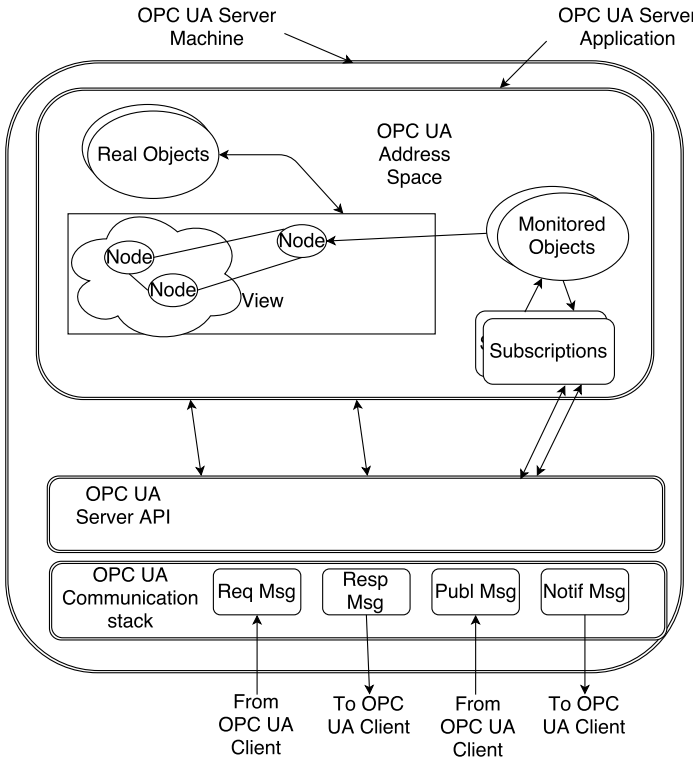

Figure 3: Internal OPC UA server structure [44]

Real objects are physical or digital items accessible to the OPC UA server application. Examples of these include physical devices [44]. I.e. an object could describe if a physical gate of a machine is open or not, and that might be required to coordinate some other mechanical operation. In figure 3, it is also possible to observe that clients can make subscriptions to servers, to get notifications regarding monitored objects. The address space instead represent the collection of information that an OPC UA server makes available to clients [44].

From figure 3 it is possible to observe that subscriptions are part of the server application at runtime. A Subscription entity is generated when a client subscribes to a server for one or more objects. After that, the server will send notifications to that client, in the form of *Publish Messages* when it detects data modifications or when an *event/alarm* gets triggered [44].

Figure 3 portraits the address space, that is composed by object nodes, that are accessible to clients through OPC UA services. The nodes in the address space represent physical and virtual objects. A view instead is a subset of address space that is of interest of a client.

OPC UA objects are of a dynamic structure (because new custom types can be defined and shared between hosts) and objects also support cross referencing amid them. To implement object references, the standard defines a node reference type, to allow objects to have relations in the address space.

Figure 4 depicts the object model in OPC UA, with variables, methods and events.

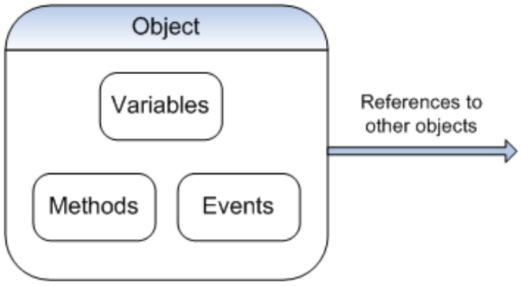

Figure 4: OPC UA object model [66]

ObjectType nodes provide types definitions, furthermore the OPC UA standard supports hierarchies of classes therefore enabling the inheritance mechanism [44]. Then, industry groups need to define their own specific information models in the OPC UA Server address space to implement their own custom protocols.

The surrounding client server architecture of the system is flexible. OPC UA servers can also be clients of other servers (servents). One possible type of interaction between the two parties are request/response exchanges. It consists in a client that send a message to a server, to perform a specific task on one or more nodes (objects) in the address space. Possible types of requests imply actions like: browsing the nodes' address space, read some node, write others, etc.

The other fundamental interaction is request/response exchange in the publisher-subscriber model. It is possible to organize servers so that redundancy of the data is achieved by having servers that subscribe to others to receive updates when the objects are modified and therefore mirroring the data [44]. But this is a just a feasible organization of the servers and the ability to organize the data flow and to program the hosts is left to whom will implement the ICS system using OPC UA.

Similarly, Peer to Peer (P2P) interactions can implement the publisher-subscriber model. In a P2P scenario, publishers send messages to *Message Oriented Middleware*, that are hardware or software systems that support the sending and receiving of messages between distributed systems [44]. There are basically two main variants of this publisher-subscriber architecture:

- A simple version, in which the *Message Oriented Middleware* is the network architecture that is able to route datagrams (routers, switches). Publisher and Subscribers then use UDP multicast to communicate [44].
- A broker based form, in which the *Message Oriented Middleware* is a broker server. Publishers and subscribers can connect to the broker and apply to a specific queue, to publish or subscribe to it [44]. The same set of services used for interaction in the client server architecture is also used in OPC UA P2P mode.

In the last architecture, the interacting parties do not know the identities of each other. There is no connection establishment between publishers and subscribers [44]. The knowledge about who subscribed to which object is left to the *Message Oriented Middleware*[44].

As previously described, the publisher-subscriber model can also coexist in a client server structure. OPC UA is a service oriented architecture because OPC UA machines send requests to use services that are running on other peer machines or servers. A client can discover which services a server is running by querying him for his *Profile*[44]. Some standard services are:

- *Discovery Service* set, that allow clients to discover OPC UA servers and also disclose their security capabilities for TLS negotiations [44]. Discovery services run on normal servers but also on dedicated ones [44]. The latter are servers that have the main goal of informing the clients on the available services on the network.
- *Secure Channel* service set; it allows to establish secure channels between clients and servers to provide confidentiality and integrity of the transmitted information [44].
- The *Session Service* set instead is used to establish application layer connection in the context of a session for a specific user [44]. It is useful to set timeout values for the session as well as for validating that a service is being access from a user that is authorized to, in a way that is also valid.
- The *Node Management* service set allows clients to add, modify, delete nodes in the address space.
- The *View* service set allows servers/clients to define subsets of the address space that is allowed to be accessed, so they can discover existing nodes and navigate between their connections to explore the structure of the actual *view* [44]. It is a service that restricts the range of operation of hosts, therefore it gives a mechanisms to manage access control of the objects in the address space.
- Alternatively, the *Query* service set, allows clients to read nodes without the need to browse them but by specifying the desired filtering criteria.
- The *Attribute* service set is used to read and write attribute values [44]. Attributes are primitive characteristics of nodes in OPC UA's address space [44].
- The *Methods* service set is defined in [46]. Clients should discover methods by browsing the available objects on the server (methods are always components of objects [44]). Then, the clients can use the service at issue to invoke the execution of a particular discovered method on the server side and return the result to the client. This mechanism is very similar to Remote Procedure Call (RPC), used in a variety of systems and protocols. Also, some methods might not be called concurrently [44].
- The *Monitored Item* service set allows clients to create and maintain monitored items. These might be: variables, attributes or event notifiers [44]; whenever one of those values change, or when there are pending alarms or events from the event notifier, a message is conveyed to the monitoring client to inform him. Monitored items also specify the interval in which the values have to be sampled for detecting modifications. E.g., a PLC client could monitor the state of a physical device e.g the state of a slider through the *monitored item* service set and decide when to take action based on the update messages provided by a server that manages the slider.
- Finally, the *Subscriber* service set is used by clients to create and manage subscriptions. Once subscribed, clients receive a notification message every periodic time lapse for each monitored item[44]. Once created, subscriptions are independent from the client server temporary session. This makes them easy to use also in a P2P model. Furthermore, to prevent non-use by client, the subscription has an expiration date, after which the client needs to renew it[44]. Subscriptions also support the recovery of lost messages; each notification message contains a sequence number that allows clients to detect if any message was missed.

Another important characteristic of the IEC 62541 standard is that it is also intended to be used as a bridge to fill the existing communication gaps between legacy technologies and modern systems [31]. A common problem that can be found in the industry is that the information flow between different levels of operation can not be easily implemented without OPC UA. With IEC 62541, domain specific information models can be adopted, in combination with secure communications, to assure seamless integration, of the different parts of the IT and OT systems [31].

Figure 5 depicts a possible industrial scenario in which OPC UA is used for interoperability.

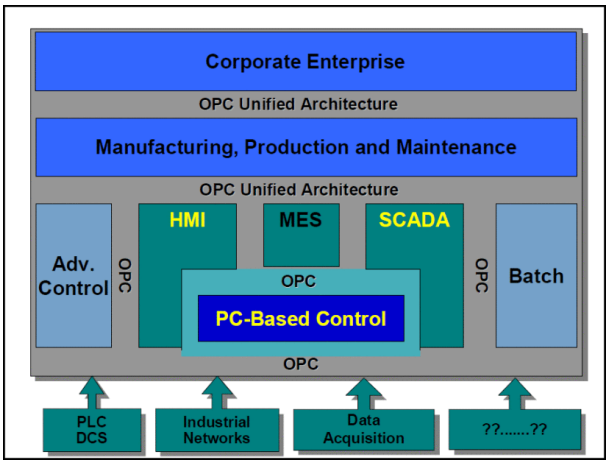

Figure 5: OPC UA provides industry standard interoperability [58]

2.1.2 OPC UA Security

Guidelines on how security should be handled during the implementation of an OPC UA specific system have been defined in [45].

The second document of the IEC 62541 standard [45] presents several attacks and the protection against them. Potential outbreak vectors are: eavesdropping, message spoofing, message reply, malformed messages, server profiling, that basically consists in an attacker probing a server to determine from his response the capabilities and configurations of it [45]. Protection against malformed messages have to be prevented by the developers that are in charge of handling security in the specific implementation [45]. Spoofing is prevented by using signatures, or some other form of authentication, on the transmitted messages. Reply attacks are prevented by having *timestamps*, *sequence numbers* and *request ID* for each message[45]. If they mismatch, a correct server/client implementation should drop the connection [45]. Once terminated, a client can re-establish a link with the server by reusing the previous session key, until it is not expired, then the client will have to establish a new session [45].

As already discussed, OPC UA client and servers make use of sessions to keep stateful connections. A likely security scenario has an attacker that tries to steal the session key of the client. This outbreak is also known as session hijacking. In an industrial scenario, a successful attack of this kind would imply that an attacker can impersonate a client (e.g a PLC) to a server machine. An attacker could therefore trigger unauthorized actions of unauthorizedly retrieve data. OPC UA prevents this by providing secrecy on data (explained later in this section). The standard supports a broad range of the most common algorithms for symmetric, asymmetric encryption and data hashing [45].

IEC 62541-2 distinguishes two disparate protocol layers that contribute to security. The classic Transport Security Layer (TLS) and the OPC UA application layer [45]. The first one implements confidentiality, integrity and availability of the data in transit, while in the OPC UA application layer are handled authentication and the OPC UA's sessions related keys (that are different from those in the TLS).

Figure 6 depicts the disparate security layers utilized to secure OPC UA.

In IEC 62541 there are different types of authentications that might take place through the application layer [45]:

- user authentication
- machine authentication
- application authentication

OPC UA applications use authentication to authenticate users to servers (while servers are authenticated to clients through pre-installed/distributes by local CA certificates). Compromising user credentials is prevented by adopting safe security policies for users' password management and strong encryption for transmitted credentials [45]. This does not take away the usual security practices employed for securing credentials such as: not writing or printing any password on papers or electronic files unless encrypted and train emloyees on malicious actors' techniques such as so-

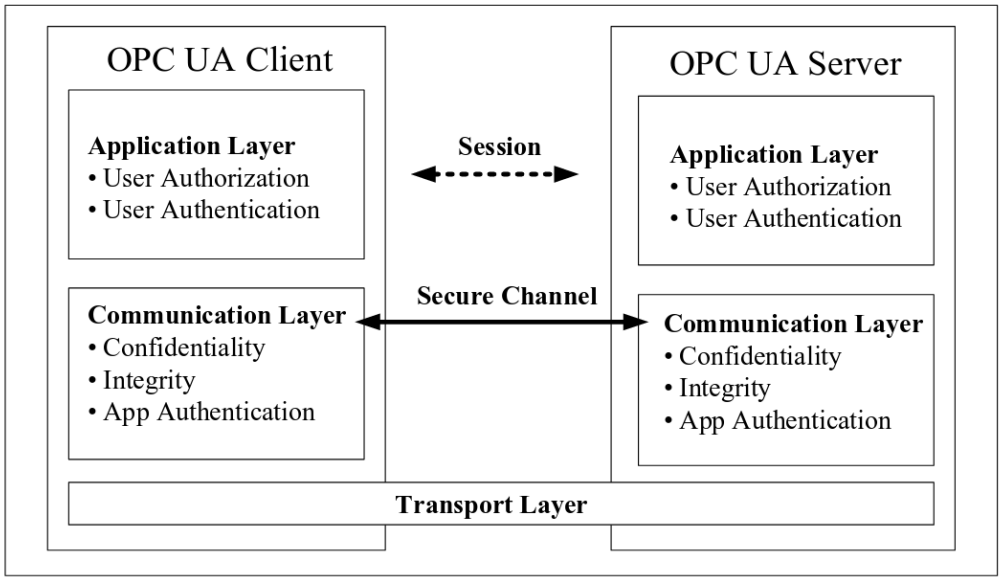

Figure 6: OPC UA security model [59]

cial engineering and how these mechanisms can be abused to gain information/control without authorization [45]. A determined attacker might even use a "password cracking" tool to try to discover credentials.

Client applications send *user identity tokens* to authenticate their users. There are three types of tokens: username/-password, an *X509.3 certificate* and a *WS-SecurityToken* [45]. If the user is using a digital certificate for authentication then he will need to win the channel-response challenge generated by the server. As usual, in this scheme, the server generate a *nonce* and provides a *signing algorithm* in response to a *CreateSession* request. The client have to prove that he is the certificate owner by signing the *nonce* using the specificied *signing algorithm* and sending it back to the server in a determined time interval [45].

X.509 certificates are the typology of certificates used to authenticate client applications to servers [45]. Different implementations in the application protocol stack might require certificates to authenticate the machine, the user and/or the application [45].

Authorizations handling for access and control to services and objects in the network is delegated to the specific OPC UA application's developers. OPC UA already provides neat mechanisms to authenticate users, machines and applications. On top of that, it is possible to build an authorization system, based on the verified identities of the applicant users. OPC UA server respond with *"Bad_UserAccessDenied"* to indicate an authorization or authentication error as specified in [47].

An attacker might also simply try to stop the servers' services by flooding it. Denial of Service (DoS) attacks are critical in ICS. If a machinery in an assembly chain stops working then the whole production chain is harmed. For availability reasons, any server, always have to define a maximum number of concurrent possible connections [45]; this might be exploited by an attacker that opens many concurrent connections to the target service and stop the original clients from getting replies as expected. Servers should send an alert to the IT administrators if the maximum number of connections is reached because then, the availability of the service is at risk [45].

Confidentiality is achieved by the use of asymmetric encryption for the key agreement, with pre-installed or safely deployed certificates from a local CA, to verify the authenticity of the server's signature. Instead, a symmetric-key algorithm is used to secure the remaining messages that are required to be sent to the host with which the secure communication has been established. Detailed encryption mechanisms are described in [48].

Integrity is verified using asymmetric signature and Message Authentication Code (MAC) verification. Asymmetrics signatures are used during the key agreement phase. On all the other messages, MACs are used [45].

Auditability is supported by OPC UA by implementing trace logs that basically annotate all the activities of all the nodes in the network. Note that the logging has to be implemented by the product producers. More details on auditing can be found in [47].

IEC 62541-2 clearly states that security is not a trivial problem to deal with and hence OPC UA software producers have to make a risk analysis to estimate potential outbreak vectors and impacts. Then they have to prevent the identified possible attacks by defining a set of security policies that will be used to prevent unauthorized access, alteration, or prevention of access to the information and services the OPC UA systems provide to the network [45]. The document at issue also acknowledge that a single layer of protection against attacks can not protect from all possible outbreaks [45]. Therefore, the OPC UA standard part 2 [45], suggests the use of "defence-in-depth" as a security strategy. This scheme suggests the use of firewalls, intrusion detection and prevention systems and control on media and devices introduced to the computer network [45]. Specific system protections might consist in: operating system configuration hardening, security patch management, anti-virus software installed and active [45]. Substantially, IEC 62541 defines security mechanisms to integrate fully customizable security policies that will have to be designed and implemented for the specific OPC UA application.

Several implementation specific security controls are specified in [45]. For example session timeouts can not be too big because if a client stays idle for a long time but without resetting the session, the server would need to keep buffered messages and other information related to him. This could lead to resource exhaustion [45]. Moreover, the OPC UA standard specifies what have to be done when parsing well formed messages on the client or the server side. But it does not include information about how malformed messages should be handled [45]. If not explicitly defined, the implementation might continue to parse OPC UA deformed messages, potentially leading to vulnerabilities [45].

IEC 62541-4 defines error codes that have to be used to notify errors to clients. But error codes might introduce information disclosure attack vectors as for example server profiling attempts. For this reason, IEC 62541-4 [47], states that a single generic error has to be returned before and at the time of establishing a secure connection. After that, appropriate error codes shoud be returned. Furthermore, part 4 of the standard also specifies that sockets have to be closed after an error message is received. Therefore, vendors should make sure that all the paths that trigger the system call of closing the network socket do not result in time based, information disclosure attacks launched by a potential network observer [45]. Finally, the standard strongly encourages implementers in taking care of arrays and strings length and to control deep recursions to evict undesired crashes [45].

Random Number Generation (RNG) is another issue related to security, especially for embedded systems. The documentation discourage to make use of the $rand()$ "C" standard function because it does not provide enough entropy [45]. As an alternative it suggests the use of Microsoft Windows Crypto library (WinCrypt library) or OpenSSL [45]. However, on embedded devices, there might still not be enough randomization. Normally, RNG functions initialize a generator based on the instant time and other pieces of local information such as: hardware IDs, screen resolution, installed software, user input, etc [45]. The trouble on embedded devices is that most of these information are missing because lacked by design. All those devices are almost identical and not as various as a desktop computer where there is more space for complexity and hence entropy. It follows that an attacker might guess an initial RNG configuration with few pieces of information because of the poor fortuity capabilities of the embedded device [45]. In addition to that, the standard suggest that security of embedded devices can be enhanced by adding hardware entropy generators when designing the device. Otherwise, certificates should not be generated on embedded devices but created elsewhere and then imported to them. For machines without entropy generators, it is helpful to store the Cryptographic Pseudo-Random Number Generator (CPRNG) on persistent memory so that every boot will not produce the same random numbers [45].

OPC UA specifies that certain functions must be only performed by *administrative* users. An example of such a delicate function are actions that manipulate certificates such as the possibility to update and revokate them. The capabilities of administrators might vary from platform to platform. Different implementations might prefer a single administrator over multiple administrators, or even having multiple layers of administrators with different capabilities is possible [45].

Multicast discovery is naturally a security issue because when a server of this type starts his activity, it broadcasts its presence on the current subnetwork so that clients can acknowledge which servers are available and build lists of them that they will later use to connect [45]. *Rogue servers* could be setup by attackers to inform of their presence and consume network and system resources, or they could also try to impersonate the genuine servers [45]. These risks can be mitigated by enabling the use of certificates for all the applications in scope.

IEC 62541 defines the concept of Global Discovery Server (GDS) to propose practical alternatives to weak multicast discovery. A GDS is a special server that provides discovery services and, eventually, certificate management capabilities for an entire plant or a whole industrial system [45]. The functionalities provided by such a type of server are several:

- Servers can register on the GDS
- Clients can query for available servers

- Clients and servers can download certificates from the GDS
- GDS can push certificates to a server
- Several GDS can cooperate to build a final list of all the available servers

The influence of a GDS is proportional to the impact of a successful breach to it. After a fruitful outbreak to a GDS, an attacker can easily subvert the whole network, or at least parts of it, because of the influence of the server. The presence of rogue GDS has to be prevented, identified and terminated as soon as possible. The GDS might be targeted because of his privileged control on the system by rogue clients and servers. If compromised, a GDS server allows an attacked to manipulate security certificates deployed to clients and servers. Furthermore, rogue GDS might induce badly configured servers to register with him and hence compromise the security certificates and therefore violate one or more of the following properties: Confidentiality, Integrity and Availability (CIA) of ongoing communications. More details about GDS security can be found at [45].

Another issue to tackle in OPC UA implementations is the parsing of special messages, such as broadcast and multi-address IP packets. Misinterpreting such special messages may lead to vulnerabilities [45]. Also, it is important to know that OPC UA specifications do not provide rate control mechanisms but a custom implementation may take care of it [45].

One more issue to face is certificate management. Normally, OPC UA applications are provisioned of *application instance certificates* to provide application level security. As already discussed, these certificates are of the X.509 type [45]. These are explained in detail in [47] and [48]. Note that certificates can be self-signed. But in that case, each application will need to have a list of trusted public keys that represent the credible certificates. On the other hand certificates can be signed by a Certificate Authority (CA) and then deployed to applications when requested [45]. The main difference between self-signed and CA signed certificates is that with the first solution, the certificates have to be deployed somehow by the developers, for example during system updates, or also by administrators that manually/remotely bind them with the specific applications [45]. With CA support instead, certificates just need to be uploaded there and the CA will take care of deploying them. The use of commercial CA companies is discouraged [45]. A local CA instead can be put in place to handle the certificate management issue. Also, applications should be capable of handling Certificate Revocation Lists (CRL) [45]. CRLs are lists of public keys that have been revoked from the system [45].

OPC UA security is a huge topic where several mistakes can be implemented on different layers of the system. Therefore, extreme attention should be dedicated to it by: designers, developers, testers and auditors employed during and after the construction of the systems. This can be achieved with the contribution of expert professionals as for example the ICS-CERT [20], specialized companies and freelancers. As written in this section, the official specifications of OPC UA, states that NIDS and other security specialized software and hardware can be put in place to enhance the level of security and monitoring of the system.

2.1.3 OPC UA implementations

There are several public implementations of IEC 62541 clients and servers [23]. These are developed by individuals as well as by the OPC foundation [33]. Fundamentally, “C++” and “Python” clients and servers are available to the public. The avilable software are mainly frameworks of the standard and to implement the specific applications, the work of OPC UA applications’ developers is required. However, for example, it is possible to use a Graphical User Interface (GUI), free software program [36], to manipulate objects on a remote server through *read*, *write* and other kind of requests. The author used the software at issue to generate OPC UA packets to enrich the capabilities of the parser developed through this document. Having traffic records files is fundamental in the development of a parser for the simple reason that without them, the testing of the parser itself results impossible. Fortunately, in this case, the standard is open and the author could use freely available implementations. Figure 7 shows the graphical user interface of the client at [36].

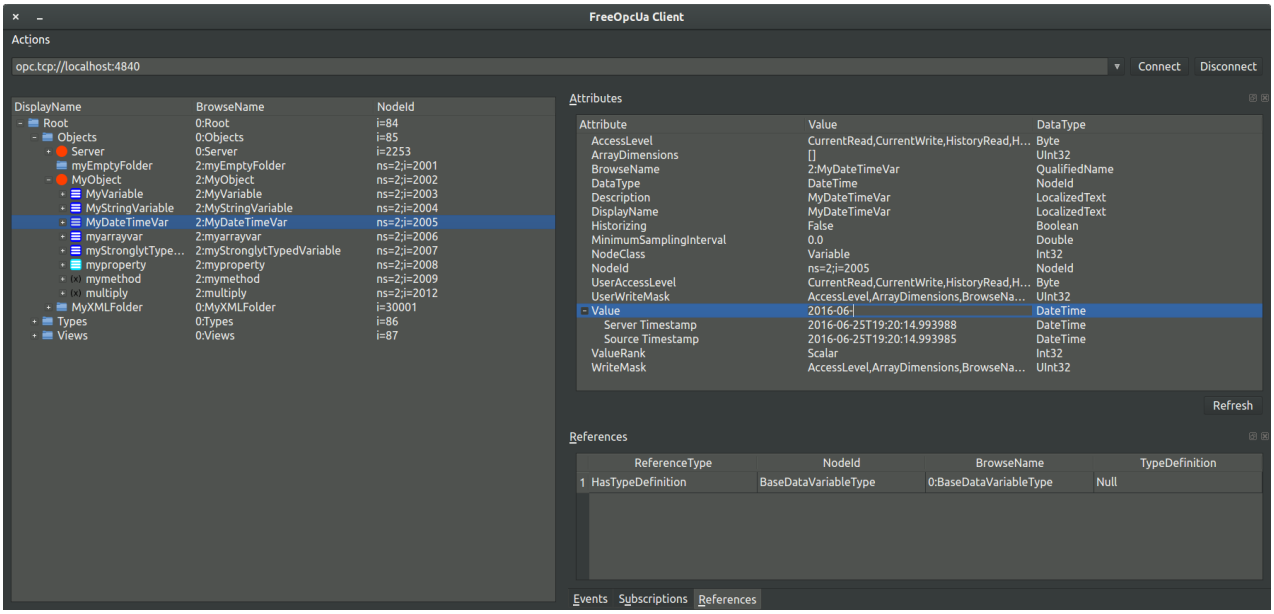

Figure 7: Sample window layout of opcua-client-gui [36]

The GUI is user friendly, the hierarchy of directories on the left shows the *objects* that the client can manipulate on the server, then the folder *types* contains new types definitions and views consist of, as the name suggests, *views*. The right panel instead encompass attributes and object references.

There are other implementations, also of libraries that support various OPC UA range of operation. However, in an industrial scenario, there would be a company that want to implement a custom OPC UA based protocol for the purpose of realizing a practical application in ICS or IoT. For example, it might be new machine for the assembly line of any product that is industrially produced. In this example, the company at issue might use OPC UA and it could employ enough developers to implement the application needed to control the machine. Therefore, the company has to deal with the development of the related sofware, furthermore, note that at this stage of development is still possible to introduce security issues, as discussed in section 2.1.2, if not enough attention is paid on the matter.

OPC UA is an open standard and this is the major reason because public official and unofficial reference implementations are available [29] [28]. Furthermore, the available implementations on the Internet will be used by companies and people that want to experiment or put in place a system based on OPC UA. By consequence, if the reference implementations have bugs, the users’ of the software might suffer because of those. As showed by the federal office for information security of the German government in the 2015, the software was actually flawed [49].

The importance of making real security risk assessments on reference implementations is a fundamental issue that is also reason of this document. Having a network intrusion detection system that supports definition of arbitrary policies on OPC UA is the goal of this project that will try to provide the presented functionalities to the maximum extent possible because of limitations of time and resources.

There is also a Wireshark dissector from the OPC foundation, that is auto-generated code by a “C#” program whose source is not public. That piece of code has been proved to contain bugs in the past [40]. Every piece of code related

to OPC UA is of incredible importance in security terms. Flaws in the reference implementations as well as bugs in Wireshark plug-ins expose their users. This is why during the evaluation process, section 3.3, it will be important to test the robustness of the generated parser to greatest possible extent.

Other implementations of OPC UA clients and servers are available on the Internet, [25] and [38] are valid examples.

2.1.4 OPC UA applications

Section 2.1 introduces the main characteristics of the IEC 62541 standard for building service oriented programs to control ICS, IoT devices and also appliances utilized in critical infrastructures. Section 2.1.2 is instead oriented at giving a description of the security features and issues that the OPC UA standard had to tackle and the mechanisms adopted to provide effective solutions. Section 2.1.3, discusses some of the available implementations of the OPC UA standard. This section is dedicated to the real world OPC UA applications in ICS, IoT and critical infrastructure devices.

OPC UA on field devices

To understand an example of context in which OPC UA has been integrated [54], the definition of field device is required. Citing [12]: "At the foundation of any process are the field devices that measure and control the flow of air, steam, water, gas or hundreds of other materials. Without proper basic setup, calibration and configuration of these devices, advanced control techniques cannot provide the levels of efficiency the technology is capable of". Substantially, the previous definition identifies field devices as critical parts of automated physical processes because they control local operations such as opening and closing of valves and switches, collection of data from sensors and raise of alerts in case the system is in danger. The paper [54] represents a research that targeted the integration of existing technologies for hardware design and implementation (the Electronic Device Descriptive Language (EDDL) and Field Device Tool (FDT)) with the OPC UA standard to achieve functional integration of field devices with the IEC 62541 standard. The authors validated the concept through the testing of a prototype that includes different field devices from process and factory automation. In conclusion, the paper at [54] states that the achieved results are important for a united future of FDT and EDDL. Furthermore, the results were presented at the Hanover fair in 2008.

Figure 8 depicts the information scheme modeled in [54] to integrate OPC UA with field devices technologies.

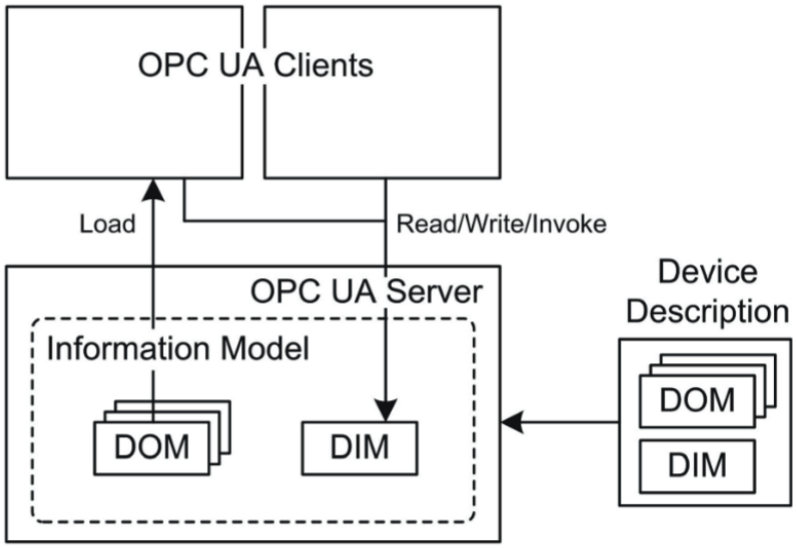

Figure 8: Architecture overview [54]

From figure 8 it is possible to note that the device description is composed by Device Information Model (DIM) and Device Operation Model (DOM). The DIM represent a way to grant standard access to the device's data, functions and higher level applications like the DOM. In this context OPC UA is applied as a technology to interconnect different field devices and make them cooperate as it is required by the specific industrial system.

OPC UA on smart grids

Another research published regarding the OPC UA integration is [51], in 2011. The paper focuses on the concept of interoperability between smart grids. The latters are considered the future towards which the current power distribution technologies are tending to, in several countries [51]. Power grids have several issues, one of which is that they must always provide enough power to satisfy all the grid's users' needs, otherwise low tension or blackouts might be experienced

by the power grid's users. The described issue implies wastage of power that is not consumed to keep the threshold high enough and therefore the need for smart grids that can improve the precision and automation with which power grids manage the actual current flow [51]. Among the contributions of [51], it is notable that the authors implemented OPC UA applications to simulate the forecasting of the power consumption of a virtual grid. The simulation has been ran on a single machine but the authors have been positive on the idea of integrating OPC UA with real world smart grids.

OPC UA on IoT devices

OPC UA has also been designed and implemented to support the IoT devices [44]. In fact, the paper at [32] discusses exactly the feasibility of integrating OPC UA with IoT devices. The document supports the idea that the scalability and the flexibility OPC UA makes it a suitable technology to support low powered, minimal resources IoT devices. Indeed, OPC UA defines a "Nano embedded device" server profile that it is intended to be used with chip level devices with very limited resources [32]. The server profile at issue is functionally equivalent to the core server and it defines OPC UA TCP binary as transport protocol [32]. The authors of [32] experimented the execution of the OPC UA server on a machine with a single chip solution, the TPS-1 (ARM9@100MHz), with less than 64KB of RAM [32].

Figure 9 depicts the architecture of the described IoT device and its components, OPC UA server included, with information on the actual memory consumption.

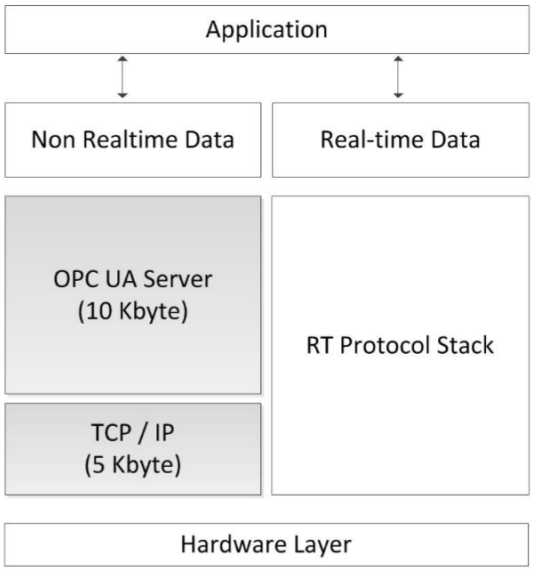

Figure 9: Implementation architecture of OPC UA on the TPS1 evaluation platform [32]

The authors of [32] concluded that OPC UA can be scaled down to IoT devices with very limited capabilities while retaining its main features. The authors successfully ran the OPC UA server written in "ANSI C" with a limited set of services (to browse the address space and read the application data) on the TPS1 system with a total RAM consumption of only 10KB [32]. The authors also state that this is probably one of the applications of OPC UA on one of the smallest servers [32]. Finally, the authors concluded that sensor devices can be used to communicate data through the Internet even with very limited capabilities thanks to OPC UA [32].

2.1.5 Summary of the BSI security analysis on OPC UA

The security analysis at issue has been organized and run by the German government to evaluate the security of the OPC UA standard (version 1.02) and of the official reference implementation of the stack written in "ANSI C" by the OPC foundation (version 1.02.344.5) [49] [24] [33].

Regarding the findings related to the OPC UA specifications, the BSI did not found critical flaws in it in fact, citing [49]: "The specification analysis performed has shown that OPC UA offers a high level of security if security Mode Sign and, above all, securityMode SignAndEncrypt is used. No systematic errors could be detected.". Furthermore, the authors of [49] discusses the fact that DoS attacks are natually very hard to be prevented in general and they can only be faced with an appropriate IT infrastructure [49]. From the analysis at issue, disparate vectors for DoS attacks has been identified. Two of those are discussed and detected by security policy scripts presented in sections 3.2.5 and 3.2.7.

The BSI added more suggestions for the OPC UA specifications that the author summarized here [49]:

- Add specific minimum requirements for random number generation
- Add notes about outdated and unsecure algorithms like SHA-1
- The specifications states to use the same key for signature and encryption (different keys should be used)
- Add support for elliptic curves (that would enforce security especially on devices with smaller processing power)
- Default values for standard parameters should be provided (differentiated for different classes of devices)
- No protection against repetitive failed authentication attempts (the server could double the wait time after each failed attempt)
- When security mode: "Sign" and "Sign and encrypt" is combined with "securityPolicyUri=None" no security layer is applied. Suggested improvement: prohibit this setting
- Arrays can be nested but no maximum limit is specified to the recursion. Specify this limit.
- Specify what should happen if a gap in the sequence numbers is identified by an OPC UA application
- Lack of forward secrecy: if an attacker discover the client and server nonces exchanged during the "SecureChannel" establishment then he can decrypt the subsequent messages. Good key exchange practices should be specified

Moreover, the BSI added statements to suggest to the OPC foundation to improve the accuracy of the specification by adding more details and in some cases they suggested to use more specific words [49].

Regarding the official reference implementation of the OPC UA stack, provided by the OPC foundation [24], the BSI reported several findings.

The static analysis of the code did not revealed any security issue. The fuzzing, developed with the *Peach* framework led to two findings: the validity of the security numbers is not validated by the stack and this lead to replay attacks (the author implemented a detection script in section 3.2.4). The other finding is cited here: "Several deviations between the specification and implementation of the OPC Foundation stack as well as inconsistencies in the specification" [49]. No other discovery has been made through fuzzing in the analysis.

The dynamic code analysis revealed memory leaks that could be exploited to compromise the availability of systems running on the stack. Further implications of these issues has not been analyzed and the BSI states that another specific analysis of each of the issues would be necessary [49]. The tests performed showed a code coverage of approximately 43% of the statements and of 23% of the branches. This because only the single-threaded server was tested during the analysis and moreover some OPC UA services such as the subscription service were not tested during the analysis [49].

The BSI analysis excluded some components from the scope that otherwise would have been too broad and would have required much more time. Such components are: the multicast DNS (mDNS) system, utilized in the discovery service set of OPC UA. The functionality related to the CRLs were also excluded from the testing.

Bro is a modern full-fledged network intrusion detection system. It is designed so that an “event engine” generates an input for a “policy script interpreter” [61]. The separation of these two components gives Bro extreme flexibility into achieving the two different purposes of parsing the raw protocols and making intrusion detection on the actual structured data. Abstracting the mechanism from the policy is fundamental for having simple definitions of both.

With the growth of the Internet and the increasing number of devices interconnected, also the amount of illicit attacks tends to increase. The activity of detecting such outbreaks is called Network Intrusion Detection[61] and that should be Bro’s goal. Unfortunately once placed a “monitor” into a network, it becomes possible to actually spy on people, other than doing network intrusion detection. Bro presents an ethical dilemma that users face when they run it. Furthermore, network monitors can be subject of attacks that aim at subverting them [61].

Bro is designed with high-speed traffic in mind. The core architecture and the binpac[64] extensions are written in C++ and compiled efficiently by the GNU Compiler Collection (GCC) for the target machine. The system reads packets directly from the network interfaces and if the analysis of such packets is too slow for the going traffic, it does queue them for later consumption [61]. Indeed, Bro has the ability to produce real-time custom notification programmed in policy scripts. They fundamentally determine how Bro will process the flowing packets. Developers can write new security policies in the Bro script programming language to define the rules of detection. Writing policies in C were not considered when designing Bro, for clear security reasons [61]. Bro scripts are written in a high-level-like language but run efficiently thanks to Bro interpreter, written in C++ [8]. Efforts are in progress to build a Bro scripts’ compiler [6].

At the moment Bro supports more than 50 protocols and several detection scripts per each one [4]. There is also a plug-in system where it is possible to add functionality to Bro NIDS, other than increasing its parsing capabilities [9]. Figure 10 shows the flow of the data from the network to the core of Bro.

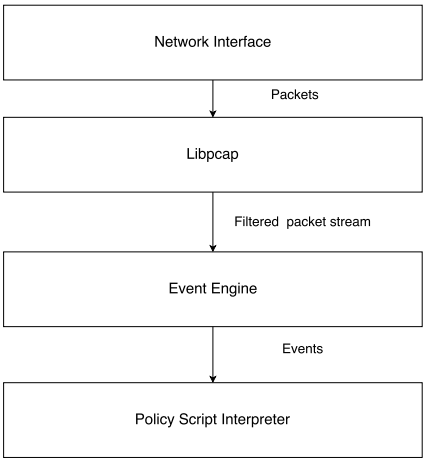

Figure 10: Bro mechanism-policy separation [61]

An assumption that has to be done when having a Bro instance monitoring the network is that the attackers know the source code of it and will use this knowledge in the attempts to overwhelm or subvert the system [61]. Mainly, three type of attacks are determined in [61]:

- Crash attacks attempt to terminate or deviate Bro code execution to respectively, Denial of Service (DoS) the NIDS, or to run arbitrary code on the target system [61].
- Overload attacks aim at consuming all the resources of the system running Bro to DoS it. A clever attacker might exploit code paths that trigger high resource usage sending small muted packets or he might even exploit memory leak flaws [61].
- Subterfuge attacks try to evade the monitoring of malicious traffic by tricking the NIDS into ignoring the attack payload. This also is generally easier when the source code of the NIDS is public [61].

For these reasons Bro have been carefully implemented to avoid this problems, but, some mistakes might still be nested in the code base.

Bro is an advanced NIDS in which it is possible to develop new modules both as components of the event engine (parsers) or as elements of the policy scripts interpreter (analyzers). The parsers are compiled into native executables and each parser plug-in is in a different static library that will be loaded by the main Bro executable. On the other hand, Bro security policy scripts are specified at runtime from the command line and they do not need to be pre-compiled into static or dynamic libraries.

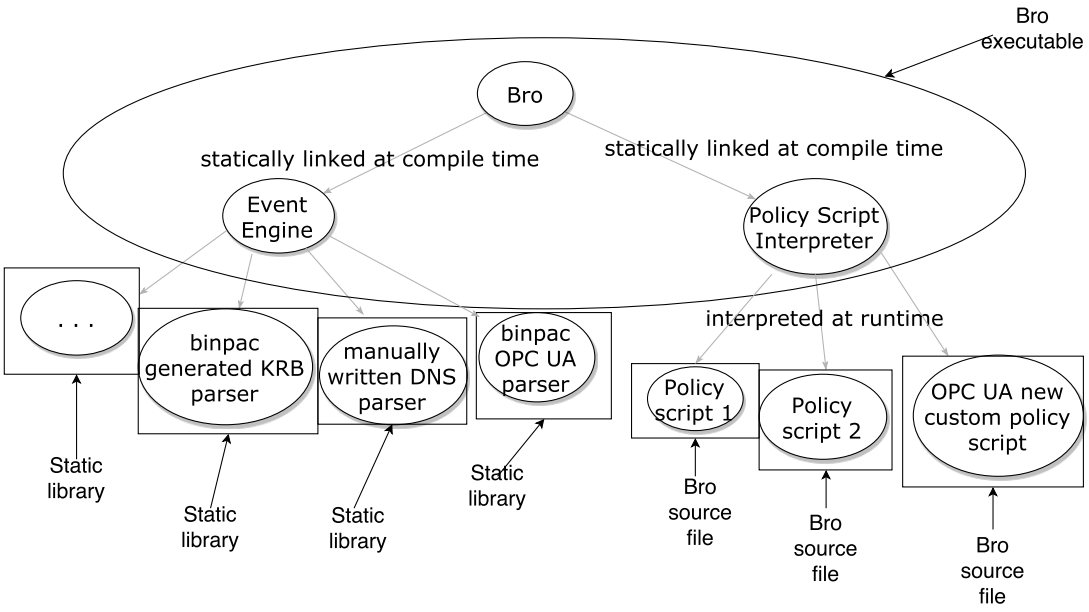

Figure 11: Bro modular architecture

It is possible to note from figure 6 that the event engine, further discussed in section 2.2.1 and the policy scripts interpreter, examined in section 2.2.2, are components that are embedded into the Bro executable at compile time. To generate static libraries for plug-ins is a functional solution as it separates the actual binaries from the main Bro executable and avoid the overhead of dynamic libraries, that are really useful when the library at issue has to be shared by several processes (not this case). Instead, policy scripts are the only software component that can be plugged at runtime but they give extreme flexibility at the behavior of Bro because they define how Bro is going to analyze the parsed traffic.

The developed plug-in gives an API interface that Bro policy scripts can use to process OPC UA traffic. The API includes new custom data structures created on purpose. Fundamentally, the plug-in defines the limits of the potentially implementable policy scripts. If there are limitations on the access to the traffic data then the Bro policy script can not do anything but the Binpac plug-in has to be improved.

Figure 12 depicts the output while compiling the Bro OPC UA plug-in to a static library:

```
Scanning dependencies of target plugin-Bro-OPCUA
make[3]: Leaving directory '/home/bortoli/tbortoli/sw/bro-2.5/build'
make[3]: Entering directory '/home/bortoli/tbortoli/sw/bro-2.5/build'
[ 42%] Building CXX object src/analyzer/protocol/opcua/CMakeFiles/plugin-Bro-OPCUA.dir/OPCUA.cc.o
[ 43%] Building CXX object src/analyzer/protocol/opcua/CMakeFiles/plugin-Bro-OPCUA.dir/Plugin.cc.o
[ 43%] Building CXX object src/analyzer/protocol/opcua/CMakeFiles/plugin-Bro-OPCUA.dir/opcua_pac.cc.o
[ 43%] Linking CXX static library libplugin-Bro-OPCUA.a
```

Figure 12: Bro plug-in compilation log

2.2.1 Event engine

As already presented, the event engine is that component of Bro that takes raw input from libpcap. It first performs checks to verify that the IP header's checksum is correct. If the Cyclic Redundancy Code (CRC) mismatch, the event engine generates an event to notify the exception and discards the packet [61]. Otherwise, Bro assembles back IP

datagrams from valid IP fragments [61]. If all the IP fragments are valid, the event engine looks up for the connection state of the relative TCP or UDP port numbers, creating a new state if none is existing [61].

Bro builds *"tcpdump"* traces from the traffic it monitors; but is the connection handler that specifies if a packet has to be recorded completely, only the headers, or not registered at all. For TCP packets, Bro's event engine tests whether the headers' checksum is genuine, then it might update the connection status if TCP packets with flags are received (SYN, ACK, RST, etc). The successive step is to generate events to feed the policy script interpreter [61]. For example when a SYN packet is received but at that point in time no connection is registered with those hosts and ports, then a *"connection_attempt"* event is generated. On the other way, if after a SYN it comes an ACK, then the event engine generates a *"conenction_established"* event. It is also possible that the remote party replies with a RST packet and at that point *"connection_rejected"* is the occurring event [61]. Similarly, when UDP packets are processed, *"udp_request"* and *"udp_reply"* events are generated. Developers can write custom protocol parsers in Binpac (section 2.3), to generate new events, expanding Bro's capabilities.

2.2.2 Policy script interpreter

When the event engine has finished to process a packet, it checks if events has been generated (they are kept in a First In First Out (FIFO) queue). If there is any, then the policy script interpreter will process such events in the order they arrived [61]. Bro's policy scripts are written using the Bro programming language with which it is possible to define event handlers that are essentially Bro functions, but with no return value [61]. Also, Bro supports asynchronous events as a link between the event engine and the policy script interpreter giving space for powerful extensibility [61]. In the next section, 2.2.3, there is a brief description of features, concepts and structures of the Bro language.

The goal of the policy script interpreter is the ultimate goal of the NIDS Bro. That is to detect intrusions in real-time or from recorded network traffic. Further goals are: to log communications, to supervise the state of the network or of the servers and all the other possible activities which require monitoring. This is the core part of Bro, the actual NIDS's policies programming. But it is only possible to program security policies in Bro when a working parser for the protocol at issue is available. In this thesis, the author develops a parser for the OPC UA standard and discuss its design and implementation respectively in section 3.1.1 and 3.2.1.

A yet undiscussed but useful functionality of Bro, is the one that allows Bro policy scripts to dynamically change the content of the flowing packets [34]. This functionality can be used to alter the content of packets and to block hosts, therefore can be used to prevent attacks. These security measures are implemented only when there is an underlying reliable and well programmed intrusion detection system because in the case in which the NIDS mistakenly drops a non malicious packet, the integrity of its connection is harmed. If that happens on OPC UA communications between industrial machineries, then the process of automation is in danger.

2.2.3 Bro language

The Bro language is the programming tool that has been developed to implement analysis of network traffic through high-level events generated by the engine discussed in section 2.2.1. The Bro language is used to implement detection and logging scripts. Potentially, any computation can be performed on a Bro policy script. They can also run system commands and arbitrarily create files on the file system.

As previously stated, Bro's designers wanted to avoid "usual mistakes" from arbitrary developers by using a strongly typed language. First-class values in Bro are IP addresses, port numbers, host names and other components that are normally part of packet's headers [61]. Bro's language atomic types are: *bool* for boolean, *int* for integers, *count* for non negative integers, *double* for double precision floating point and *string* for byte sequences [61]. All of these types but *string* are called arithmetic types and mixing them in expressions automatically cast *bool* into *count*, *count* into *int* and *int* into *double* [61] . Other types are: *time*, *interval*, *port*, *addr*, for IP addresses and many more. Bro also supports different aggregated types: a *record* is an element that contains a collection of items of arbitrary types [61]. *Table* is another aggregated type: it allows to link keys to values in an associative array style [61]. Finally, *Set* is the type of a data structure that can contain an arbitrary number of elements, similarly to normal lists.

Bro provide several C-like operators to manipulate atomic types. To access the attribute of a record field the *"$"* symbol must be used. Assignments of aggregate values only make the assigned variable to point to the assignee [61], so that efficiency is not wasted on huge objects' copies. Bro's developers reserved to the future the possibility of introducing a *copy* construct, to make "deep" copies [61]. The *"in"* operator can be instead used to check a string against regular expressions or if a key is contained in a table or in a set [61].

Finally, Bro provides a set of functions to perform operations like: *"fmt"* to format a string in *"sprintf"* style, *"mask_addr"* applies a network mask to a certain network address, *"open"* and *"close"* are used to manipulate files, while *"getenv"* gives access to the environment variables and *"system"* executed a string in a Unix shell command and others [61].

Variables in Bro have two possible scopes: local, to a function or event handler, or global [61]. Local variables are declared using the keyword *"local"* [61] inside functions or event handlers. Global variables are preceded by the *"global"* keyword and must be declare out of any defined function body [61]. *"const"* is used to define constants.

Advanced functionalities like implementing tables that use developer's defined data structures is unfortunately not supported yet but the Bro project is still in development and therefore, new versions of Bro will probably overcome gaps like this one.

In conclusion, the described language is Turing complete and therefore the right tool to achieve stateful intrusion detection analysis [7]. Bro language provides handy types and functions to policy script writers [61]. Strong type checking and exception handling prevent typical programming errors [61]. These and all the other described features are what make Bro the ideal candidate as a NIDS.

2.3 Binpac

2.3.1 Overview

A key problem in network analysis is the parsing of packets [64]. Protocols have always been the mean of communication between different machines. Protocols have been evolving with computer networks and software. New applications rarely implements the network protocol for scratch but reuses the already existing ones. Some other times for efficiency reasons, or to add functionalities, developers rewrite simple protocols from scratch to make them fit the purpose of their own applications. The specifications of these protocols might be public through Request For Comments (RFC), other kind of documentation, or they might be private, when the protocol is developed by a company that does not release the details of its structure to the public it is then called, proprietary protocol. Without the original specifications, understanding syntax and semantics of a protocol is really hard because it requires reverse engineering, that process of understanding how something works without having the inner structure and details of it but a black box where to put input and get output.

Binpac has been developed to solve the problem of parsing network application protocols generically, so that an efficient and robust parser can be produced from Binpac and embedded in any software [64]. The process of parsing involves transforming a shapeless bit stream into a typed, structured, high-level representation of the traffic that can then be used for semantic analysis of the network interactions [64]. Binpac has been developed as a declarative language. The authors of [64], show how parsers for common network application protocols can be built and show their robustness and efficiency through testing. Then, they use this new parsers to substitute several handcrafted parsers implemented in Bro, to show that Binpac can effectively be easily combined with other software to enhance its functionality and abstract the network traffic parsing problem [64].

In short, Binpac is a compiler for his declarative language. It is similar to Yet Another Compiler Compiler (YACC) because of its declarative nature. They both generate parsers and they both run on well defined, formal structures (although of different kinds). Binpac source files are at their basic form just declarations of the packet structures, with the possibility of adding "C/C++" code in line to enhance functionalities [64]. The authors of [64] observe that it is generically hard to write efficient and robust application-level parsers without implementing several mistakes and repeating a lot of code because ot the nature of parsing itself. It requires robust and fast software to have reliable parsing and that is not trivial of the complexity of modern protocols. Furthermore, application layer parsers are important components of a lot of software that is used for the most disparate purposes. Monitoring tools such as *tcpdump* and *Wireshark* make use of application level parsers and real-time network intrusion detection systems such as *Bro* and *Snort* do exactly the same [64].

Binpac is therefore an improvement from the point of view of the developer that has to implement the support for a new protocol for some software but is also an improvement in terms of robustness of the parser itself and hence an advancement in the security of the parsing routines themselves. It is quite tragic, to think that security enhancing tools such as monitors and NIDS might have flaws that allow an attacker to take over the monitoring system [42]. But it is normal, computers are inherently complex machines and it is hard to foresee their behavior in any condition. Furthermore, while for software bugs there is a logical explanation (sometimes, unfortunally twisted up), the hardware bugs involve more physical phenomena other than pure logical issues. When the explanation requires notions of quantum physics, not many people may have clear ideas about the issue. Another curious phenomenon is computer malfunctioning provoked by cosmic rays [22] and the row hammer bug [55]. For all of these reasons, computers must be carefully programmed and configured, with a continuous cyclic work of improving, studying and testing that better suit the development process of such complex, sometimes, apparently unpredictable machines.

The authors of Binpac analyzed the modern structure of application level protocols to gather the knowledge needed to develop a tool to parse them all. One of the main differences that the authors identified in computer protocols is that they are categorized into two classes: *binary* and *ASCII* protocols [64]. For example, messages of the first class, such as the Domain Name System (DNS) requests and replies consist in a number of data *fields*. These fields, express their semantic values by standard types such as integers and strings [64]. On the other hand, human readable ASCII protocols, such as the Hyper Text Transfer Protocol (HTTP) and the Simple Mail Transfer Protocol (SMTP) have that their requests/replies are separated by carriage-return/line-feed special characters. Another observed characteristic of modern application layer protocols is that they use variable length arrays to convey non fixed magnitude of information [64]. Strings are simply array of characters. Binary protocols instead use specific encoding scheme to represent the data contained. They might use the classical little endian or big endian encoding, as well as other, more sophisticated techniques.

One of the biggest problem of real-time protocol parsers is that they need to handle concurrent input [64]. In a real network scenario it is normal that multiple parallel connections are established and exchange data concurrently. Parsers generated by yacc/lex process input in a serial fashion. When the syntactic/lexical definition is terminated, the parser generator run to generate "C" code that will parse the defined grammar/syntax. Because there is only one input per time, in the compiler, only one thread is needed. More threads can be used to exploit the capabilities of multiprocessor platforms. But, a network parser requires more threads, to handle concurrent packet flows, or a single thread that process input incrementally as it arrives and switches between different flows to handle all of them.

As already outlined, robustness of network protocol parsers determine the security of the systems on which they run on. An attacker may craft special packets to lead the protocol parser to an error. Errors may be provoked by irregularities in the network traffic or corner cases that were not included in the specifications of the protocol and their consequences might vary from infinite loops stalls to program crashes that might eventually be controlled by a keen attacker to achieve arbitrary code execution. Moreover, contrary to compilers, network protocol parsers can not just complain and terminate if the input is malformed or incomplete; they have to handle it robustly and consistently generate message logs to notice the unusual event to the user, programmer or administrator that handle the software that involve the parsing.

2.3.2 Language

Binpac defines a declarative language that must be used to determine the rules that will drive the generation of the application protocol parser that will be the output of the compilation process. The generated parser will be in "C++" to be efficient, flexible and object oriented [64].

A Binpac type describes both: the layout and ordering of the bits and the structure that will be built from them after parsing [64]. Elementary types in Binpac are: *"int8"*, *"int16"*, *"int32"*, for signed integers, while: *"uint8"*, *"uint16"*, *"uint32"* represent unsigned fixed length integers [64]. A string can be described by a *constant string*, a *regular expression*, or a more generic *bytestring* that can be of a variable length, *"&length"* or long till the end of the data [64].

External types can be defined by writing "C++" code. Those types will be valid when used in computations (e.g as parameters types) but they can not be used as types of any data fields in messages of the defined protocol. User can define composite types: using *"record"* to describe a collection of heterogeneous variables, *"case"* that is a conditional statement to evaluate the type of a field and *"array"* that is a sequence of elements of the same type [64]. For each user defined type, Binpac generates a class with fields as class members and a parse function to extract the fields from a sequence of bytes based on the described layout [64].

Byte order is another property of the type and packets structures defined by the developer. Using the attribute *"&byteorder"* permits to specify one of them. Another important feature of the language are derivative fields. They are used to keep intermediate computation results, or to further process the outcome of parsing. Derivative fields are defined within *"&let{ ... }"* attributes [64].

State management is a fundamental feature offered by the Binpac language. It defines the notions of *flow* and *connection*. A *flow* is just a sequence of messages, while a *connection* is a pair of flows. "Connection" means any type of network flow between two hosts [64]. Moreover, each *flow* might be of two different input types: *"datagram"* if the messages are received in one packet or a *"byte stream"* in which the data delivered boundaries do not match the message confines. The two input types can be specified using *"datagram"* and *"flowunit"* keywords respectively.

Although the mechanism of types' parametrization allows to keep temporary information within a message, it might be possible that in a certain scenario, the developer needs to store information at connection or flow level. To declare a parser that will be capable of having a per connection/flow state, the following syntax must be used when declaring the analyzer: "*analyzer <name> withcontext*", at the beginning of the root Binpac source file [64].

In yacc, the developer can inject "C/C++" code next to the grammatical productions to process for example the construction of abstract syntax trees that represent the parsed program. In Binpac there is a different approach; the source code is translated into "C++" by the compiler and then the parsed data are further passed to the Bro policy scripts by generating events. However, types can be enriched so that, it is possible to link a call to an external function that is fired when an instance of that type is successfully parsed. Therefore making it possible to connect "C++" code to events of successful (or not) parsing. Note that this kind of connection is the only link between Binpac and Bro [64]. This thesis deepens the argument subsequently, especially in the implementation section 3.2.1. In the same is also discussed the exception handling for malformed packet detection.

Binpac implements simple syntactic constructs that permit to easily separate parsing from analysis code; it is similar to a high-level, low-level tasks separation. In Binpac it is possible to specify packets' headers definitions separately from the event generation to pass to Bro. Instead, semantically, the two separated pieces of code will combine together by first parsing the packet and then generating events in sequence. Therefore, Binpac implements some "separation of concerns", hence making its code cleaner by separating parsing and analysis sources [64].

As previously stated, the Binpac compiler supports incremental input. This mean that it can keep track of the state at which the parsing is arrived to. *ASCII* protocols are usually line based or alternate between explicit denoted length and line based. Thus, Binpac provides two special attributes: *&length*, used to state the length of a field. The other is *&oneline* that basically instructs the parser to keep reading the input till the End Of the Line (EOL) [64].

Binpac detects and catch various kind of failures by throwing run-time exceptions. It is possible then to handle the errors, to recover, or at least, report the exceptions. As already discussed, failures might arise from incomplete or incorrect specifications in the Binpac code, missing parsing cases; malformed packets that contain wrong length field values, that are a possible scenario in case of actual cyber attacks [17][16]. Another source of errors are interrupted inputs that leave the parser in an inconsistent state [64].

Boundary checking is that process of verifying whether the input byte stream fits within the input buffer, as well as controlling that the tasks performed on it do not imply read or write operations out of the programmed buffers. Errors on boundary checking lead to buffer overflows vulnerabilities that afflict computer systems since their existence. This process can be made efficient by logically grouping the checks together as much as possible and put them in the code routine every time that check is needed. Binpac does implement this kind of simplification [64]. The code of parsers is very arithmetical, repetitive and mechanical in general, that is why a simplification as Binpac comes very useful.

As previously introduced, handling chunked packets is something a parser has to deal with. Binpac specifies a *"&chunked"* property that can be bound to byte sequences. In this way, the parser will pass to the *"&processchunk"* function that can simply skip over the gap if there is one. In this way Binpac handles most of the common content gaps. Instead, in hand-written parsers in Bro, packet gaps are handled in a similar way but on an individual case basis. The solution provided by Binpac handles this class of problems universally for all protocols [64].

Binpac also enforces type safety. This task is accomplished by generating parsing code that ensures that only the case selected during the parsing can be accessed. If this is not possible, it throws a run-time exception. Also, access to elements of arrays is always enforced by boundary checking [64]. Last but not least, Binpac allows embedding of "C++" code and hence it can not guarantee complete safety during the run-time because errors might be introduced arbitrarily.

Another notable feature is the opportunity to have user-defined error detection functions that are automatically bound by the parser compiler. This functionality is achieved by the use of the *"&check"* attribute in the specific field that the user wants to check [64].

Errors handling is currently implemented in Binpac as follows: when a failure in encountered, it logs the error, discard the unfinished message and then proceeds to the next chunk of data. A potential limitation of this approach is that, for stream based protocols, the successive packet might not be aligned with the next payload chunk [64]. In the future, the authors of Binpac plan to implement a mechanism to rediscover message boundaries in such corner cases [64].

2.3.3 Evaluation

Binpac is currently used in Bro to parse 25 of the 52 supported protocols [4]. Between these there are: *ssh*, *sip*, *krb*, *imap*, *mysql*, *modbus*, etc. Furthermore, the authors of Binpac have compared the complexity and performances of the two different approaches: hand-written Bro's protocol parsers and those auto-generated from the specifications by Binpac for the HTTP and the DNS protocols [64]. The following table report the comparison.

Protocol	Hand-written			binpac		
	LOC	CPU Time (seconds)	Throughput	LOC	CPU Time (seconds)	Throughput
HTTP	1,896	538–541	244 Mbps / 36.7 Kpps	676	442–444	298 Mbps / 44.7 Kpps
DNS	1,425	37.3–37.5	18.6 Mbps / 13.3 Kpps	698	44.7–44.8	15.6 Mbps / 11.1 Kpps

Figure 13: Binpac versus hand-written protocol parsers [64]

From the previous table it is possible to observe that Binpac parsers are much more efficient in parsing the HTTP compared to the hand-written ones. Instead, the DNS protocol is parsed faster by the hand-written parser. The previous observation suggests that Binpac's generated parsers are more efficient in handling *ASCII* based instead of *binary* based protocols. However, the number of lines of code decreases drastically in binpac. The said change shrinks the overall complexity of the parser, simplifying testing and code reviews that will eventually be performed on that software. Therefore, Binpac provides a comprehensive new approach to the problem of parsing that improves security and usability of the state of the art of application protocol parsers.

3 Contributions

This section contains the report of the contributions of the project. Sections 3.1.1 and 3.1.2 treat the design of respectively, a binpac plug-in and Bro's security policy scripts. Instead, sections 3.2.1 and 3.2.3 discuss the implementation of the binpac plug-in and Bro policy scripts. Sections 3.2.4, 3.2.5, 3.2.6 and 3.2.7 describe the implementations of policy scripts for attacks identified in [49]. Furthermore, in section 3.3, the plug-in is tested for performances and robustness.

3.1 Design

3.1.1 OPC UA binpac

Bro allows developers to write plug-ins to expand the available functionalities and to support new protocols (section 2.2). The author used this feature to implement the parsing of OPC UA in the NIDS Bro. Every plug-in in Bro has basically two main directories. In one are defined all the Binpac source files. In the other the developer can write default policy scripts that will be loaded with the plug-in itself to support features such as writing what happens to log files. The default policy scripts embedded and deployed with the plug-in are not limited in terms of complexity, therefore the user can load all the scripts he likes till resource exhaustion.

The Binpac source folder is located under "*$BRO_PATH/src/analyzer/protocols/$PROTOCOL_NAME*" while the Bro policy script default directory is under "*$BRO_PATH/scripts/base/protocols/$PROTOCOL_NAME*". Those, are the main locations of the source files written in Binpac and Bro (*.pac and *.bro).

Another important file is "*$BRO_PATH/scripts/base/init-bare.bro*" because it runs at every instantiation of the Bro's policy script runtime environment. The aforesaid file will be used to load the custom types for the OPC UA framework.

To implement the parser the author wrote Binpac code to specify the logic structure of the possible OPC UA packets. In Binpac, every packet is parsed by starting from a non-terminal symbol, "*OPCUA_PDU*". Then, based on the "*message_type*", Binpac will choose which structure it has to parse. Non OPC UA packets will not result in successful parsing because the grammatical definition will not match on them. Once the packets format is defined, it becomes possible to generate Bro events that will be passed to policy scripts for further processing (section 2.2.1). This abstraction allows to completely separate the problem of parsing from the application of security policies on the traffic. Binpac code handles parsing, while Bro code implements the analysis. Furthermore, meaningful data structures like vectors, sets and records can be passed to Bro scripts with events.

This system is efficient because the Binpac parser is converted to "C++" from its compiler and then to machine code from GCC. Even if the policy script engine runs as an interpreter it is efficient because of the language simplicity and its specificity. Moreover, the created Binpac plug-in will notify events in real-time to the Bro's policy engine that will consume them before they fulfill all the available memory. As discussed in section 2.2, Bro is a network intrusion detection system that consumes traffic in real-time by keeping the received information in a FIFO queue so that it will be processed in chronological order by the policy script interpreter. If Bro can not keep the pace with the network traffic, it will eventually fill all the available Random Access Memory (RAM) and the system would start trashing between RAM and swap storage space, tragically slowing down performances and responsiveness of the whole system. Therefore, it is important, in production scenarios, to test that a certain system, with a specific configuration, programmed with some policy scripts, can keep the pace of a certain network without exhausting the resources. Otherwise, running Bro on a cluster architecture might bring the missing computational power to analyze all the traffic [5].

Binpac is a declarative based language discussed in section 2.3. It has compact and accurate structure definitions for fine-grained packet description. It allows also to bind callback functions and variables to attach metadata with the parsed structures. In the implementation there will be a heavy use of Binpac to define the different packet structure of OPC UA packets.

Finally, regarding the organization of the sources: the files are organized hierarchically having that the sources that contain the principal data structures include the needed substructures form other minor files. This is a modular organization of the sources that aims at keeping the content of each file of a limited length, possibly printable in just one screen rendering and divide the content based on relevancy. To clarify, i.e. "opcua-message.pac" does define the binary substructure of a "MSG" typed message. But it also include several other files to allow definitions of other files to be used locally. Fundamentally, the include mechanism works as the basic "C++" *include* pre-processsor directive.

The Binpac plug-in is probably the most substantial contribution of this project because it opens the Bro network intrusion detection system to the analysis of OPC UA traffic. Once implemented and tested, the plug-in can be used to write policy

scripts to detect violations of the standard, malformed packets and different types of attacks. Furthermore, policy scripts can also be used to implement arbitrary analysis on the structured information flow, with the help of files, databases and all the informational constructs that might be necessary.

As design issues the author did not needed to model complex schemes but to organize the parser structure in small, easy to read files with meaningful names. Plug-ins written in Binpac are very compact because of the synthetic nature of the Binpac declarative language. A plug-in structure is normally of this form:

- Binpac parser definitions
- Binpac events and types definitions
- build and pass events to Bro scripts

The previous list can also be rewritten as follows:

- packets' structure ".pac"
- events and types ".bif"
- data conversion and events generation ".pac" (containing mainly "C++" code)

These are the main issues that a developer has to face when implementing a Binpac plug-in to parse an unsupported protocol.

Figure 14 depicts an overall picture of the pluggable components of Bro. It is possible to observe that there are plug-ins, written in Binpac, which purpose it to extend Bro's parsing capabilities to unsupported protocols. There are also default Bro policy scripts, as previously discussesd in this section these are policy scripts that are loaded by default in Bro. Furthermore, the system includes a file that contains the definitions of the functions and data structure utilized in Bro's policy scripts. Finally, it is also possible to provide additional security policy scripts through the command line.

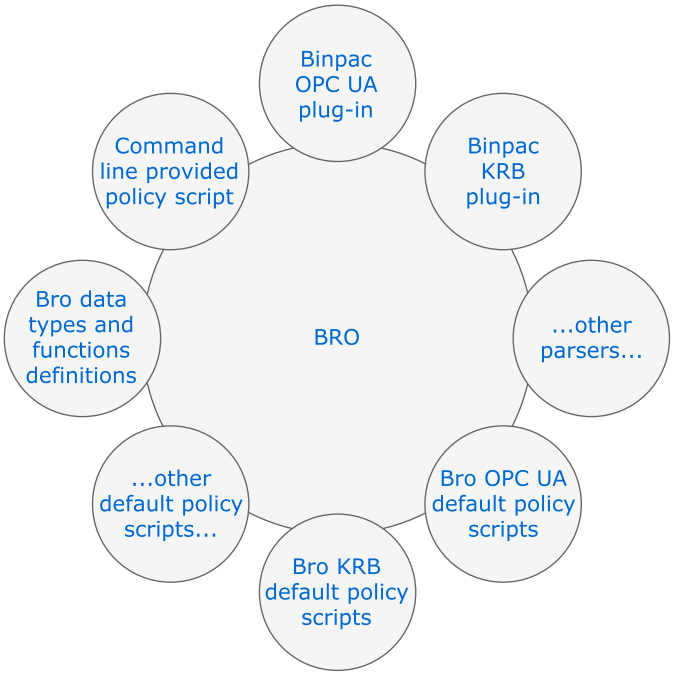

Figure 14: Bro components

A big advantage of Binpac is that the developer does not have to care about the low-level classic issues of implementing a parser. The programmer can focus on giving meaningful protocol descriptions and write code to build and fill structures that will be passed to the Bro policy scripts for further processing.

Once the parser is working, the efforts can move to the development of effective policy scripts that can detect something interesting like attacks, malformed packets and violations of the standard.

In section 3.2.8, this thesis describes that the parser also provides the printing of the exceptions, that is useful to detect malformed packets. The aforesaid transient data might trigger a memory corruption flaw or a similar one in some OPC UA reference implementation. It is useful that the parser notices of corrupted packets, therefore it is possible to use the system to detect attempts of attack, as well as deviations from the specifications. Another possible application is to use a random bit flipping program or a similar mutative algorithm on OPC UA sample packets and then pass them to the parser to detect malformed packets and build specific test cases to use against some reference implementation.

Some source files, are of the extension ".bif", that stands for Builtin Functions (BiF). BiFs sources provide a convenient way of interacting between the "C++" event engine and the script policy processor by making "C++" functions available to policy scripts and by making Bro variables, types and constants available to the "C++" [11]. Examples of builtin functions' files in the plug-in are "events.bif" and "types.bif". They declare the API (events) and the types, respectively.

Bro variables and constants are represented in the "C++" layer as pointers to instances of the *"Val"* class. Furthermore, there is a hierarchy of classes in Bro that builds the basic type environment for the security policy script interpreter [11].

Bro types are instead represented in the native "C++" code with the usage of the *"BroType"* class. Again, classes are organize on hierarchies, in fact, a Bro *"record"* is identified by the class *"RecordType"*, that inherits methods and attributes from the class *"BroType"*.

Similarly, Bro events are represented as instances of the *"EventHandlerPtr"* class. Furthermore, BiF files support namespaces and modules usage. Using the same syntax as in Bro's policy scripts, *"module <MODULENAME>"*, it is possible to specify the declaration of variables, types, enumerations, constants, functions and events belong to a specific module. If using *"GLOBAL"* as a module name, then the scope will be the default one.

To write the BiF sources is fundamental to have the plug-in running. Substantially, BiF scripts are the connection layer that links the "C++" parser to the "C++" policy script interpreter. They must at least declare the events that will be generated. When Binpac compiles BiF sources, it generates the required "C++" classes, and therefore the reference of it from other parts of the Binpac source, for example when the event has to be fired, there is a call to the event function that has not been declared directly by the developer but generated by the Binpac compiler before compiling the rest of the source.

In this part of development, during the parser creation, the author used the GNU is Not Unix (GNU) Debugger (GDB) to debug Bro and to understand the mistakes committed in order to fix them [13]. GDB was used with the graphical front-end called "DDD" [14]. That is a graphic user interface for command-line based debuggers like GDB. It supports memory inspection features, monitoring variables, breakpoints, various stepping modes and other features similar to modern symbol based debuggers.

3.1.2 OPC UA policy scripts

Bro is a high-level scripting language designed to write policy scripts (section 2.2.2) for the Bro NIDS. As previously stated, it was implemented to give a fast, easy to use and secure programming tool for the network intrusion detection system's policy definitions. It is Turing complete and very similar to "C++", in fact it can basically embed "C++" code. It supports user defined data structures, as well as several default types. Bro language is the tool that the author used to implement security policies.

Having a working parser for the OPC UA traffic makes it possible to apply policy scripts in the Bro language. The author programmed the parser in Binpac to solve the parsing problem, as discussed in section 3.2.1. This section is dedicated to define how policy scripts should behave in terms of general guidelines.

Security policies can be very broad in general. A potential analysis may inquiry the integrity of the packet by checking for invalid values, checksums, etc. Also, the variety of security policies should cover the interactions the analysis wants to cover. The standard is very wide and complex, a thoroughful analysis of all the interactions would require huge efforts. Furthermore, the encypted traffic can not be revealed because of Public Key Infrastructure (PKI) presence [45], unless some "SSL strip" mechanism is in place. However, it is still possible to analyze SSL handshakes and all the unencrypted

traffic without breaking the encryption. Basically, real-time data processing of the OPC UA packets is possible within the limit of sniffing traffic on a network, or providing packet capture files.

The tool used in this thesis to implement security policies is the Turing complete Bro language. Its structure, as previously explained is similar to the usual "C++" with no preprocessor macros. The code can be written in functions, the developer can define custom structures and variables made of complex and simple types, for example: *counter*, *set*, *table*, *string*, etc.

The main interface between the policy script and the event engine are, as the name suggest, events. As described in section 3.1.1, events are defined in BiF files, as well as types. Events are the APIs that bring structured data to the policy script runtime environment. Bro developers that wish to write an analysis for some protocol have to first take a look at the available API. Then, they can organize their operations based on them. The advantage of this system is that in this way Bro code just focus on the analysis of the structured data instead of having to care for the parsing, that is accomplished by the plug-in Binpac code.

Bro policy scripts use an event based programming paradigm that comes very handy as well as all the other domain specific features they implemented to make it a successful tool to write security policies scripts.

There is not that much which need to be designed when writing Bro policies scripts. During the development, the harder part is probably the writing of the plug-in that provides a framework to Bro's policies scripts because it is lower level programming and it is harder to debug. Once the development moves to policy scripts, then the programming comes easier, because the Bro language is of a higher level compared to Binpac and "C++". It has constructs that simpify tasks that otherwise would require more complex definitions. It does not need the developer to allocate/deallocate memory or to take care of bounds checking. It provides specific keywords to handle particular tasks like resetting the value of a variable after a certain amount of time it was not read and/or written. The Bro language is the fast and secure solution that the Bro project proposed to make NIDS specific processing. The next sections describe how this language can actually be used to implement functional detection mechanisms.

3.2 Implementation

The author implemented the solution using principally three programming languages. Bro, a "C++" like language, to write stateful security policies processing; Binpac, a declarative based programming language created with the only purpose of compiling network application protocol parsers by writing formal specifications of the packets. Also, "C++" code has been written by the author to complement Binpac functionalities and to manage data conversion between Binpac and Bro.

The code has been written following a simple programming style. Names of user defined functions usually describe what they actually do. Conversion routines instead take the name of the type that is going to be converted to the Bro policy script Runtime Environment (RE). Names of types in Binpac are all uppercase letters. Name of Bro's types all lowercase characters. Using this basic distinctions the author was allowed to give the same names to Binpac's types and those of Bro. Precautions like those just described improve the source code readability. Names of fields in binary structures have been taken from the official documentation (IEC 62541) and from the Wireshark OPC UA dissector [41] to respect as much as possible existing conventions and to not foster confusion.

With the final solution is possible to feed Bro with policy scripts that other developers may write and exploit the new parsing capability implemented with the OPC UA plug-in. The author fundamentally provided an extension to the Bro parser and event engine to provide a new set of events, that will function as Application Program Interface (API) for policy scripts at runtime. Refer to appendix D for detailed informations.

The contributions are not limited to that because with the solution, a basic set of policy modules have been provided. To achieve tasks such as: logging OPC UA network packets information content and enforce security policies. Potentially arbitrary policies can be defined thanks to the stateful interpreter processor. Moreover, detection of malformed packets is implemented in the OPC UA parser, section 3.2.8.

Fundamental part of the contributions is also the script that the author provides to compile and embed in Bro the plug-in. The author did not used any Integrated Development Environment (IDE) to develop the project and hence all the compilation, linking and testing processes were done through a standard *"shell bash"* and other FOSS software.

3.2.1 Binpac

The core of the plug-in is the Binpac parser. Its root file is "*opcua.pac*". It defines basic and mandatory Binpac's structures to define elementary objects such as the data flow and include the rest of the sources. In the source code it is possible to observe that there is a statement that defines a "*flowunit*" named variable. That is needed to make sure that Binpac uses incremental input and allows us to use special keyword such as "*&length*" and "*&oneline*" in the source code. If instead it would be defined, "*datagram*", Binpac would not enable this features. In the source code is also stated that without incremental input the parser would be more efficient but the author needs to define variable length strings and hence, "*&length*", is needed.

In the main source file, *"opcua.pac"* are included also *"types.bif.h"* and *"events.bif.h"*. Those are "C++" source files, generate by Binpac using *"types.bif"* and *"events.bif"* respectively. They define the added Bro structures (types) and API (events).

A very important source file is *"opcua-protocol.pac"*. It defines the *"OPCUA_PDU"* type, that is used to initiate the definition for all the packets. It is defined as a *"record"* and it contains the representing structure of information written using terminal and non-terminal symbols. As previously described, in section 2.3, there are several default types: *"uint8"* to *"uint64"*, and *"bytestring"* for variable length strings.

A fundamental construct is the *case* statement. Its syntax is the following: between round brackets, right after the *case* keyword is placed the field or expression using the available fields, parameters and/or metadata. The evaluated expression will be compared sequentially to the different cases and the matching one will pass. Furthermore, the "*default*" value can be used to match any value, it is practically a wildcard. This type of constructs might remember of functional programming, when the functions are all defined on a simple case basis. Also, the dynamic typing just described allows also to pass fields from the current structure to the dynamically defined type as Binpac supports this mecchanism. Therefore, information can flow from the Protocol Data Unit (PDU) to the contained message. Following, an example of the *case* construct from the plug-in's Binpac source:

```
type QUALIFIED_NAME() = record{
        id: uint16;
        len: uint32 &byteorder=littleendian;
        val: case(len) of{
                0xFFFFFFFF -> e: empty;
                0x00000000 -> f: empty;
                default    -> name: bytestring &length=len;
        };
}
```

The precedent code listing describes a "QUALIFIED_NAME", that is a specific data structure of OPC UA [48]. The logic of the case statement says that if "len" is different from zero and "0xFFFFFFFF", the parser will seek for a string of length "len", otherwise it will assure "len" is the last field of the structure.

Another important feature of Binpac is that the keyword *"&let"* allows to define dynamic variables for metadata definition. It is possible to add them by using a graph brackets' block, after the basic field definition. Refer to *"opcua-protocol.pac"* for a concrete example.

Important functionality of Binpac is the possibility to specify the byte ordering to simple or complex types. Two basic byte orders that the author used are the classical *"bigendian"* and *"littleendian"*, but for OPC UA, basically everything is in *"littleendian"*.

The *"&length"* keyword lets you define the length of a *"bytestring"* or of a *"record"*. In the OPC UA standard, the field *"length"* always precede a string to inform of its length [48]. More often, strings are terminated with a "NULL" or "CRLF" in network protocols. In the case a "CRLF" is used as terminator, the attribute *"&oneline"* can be used to describe the dynamic extent of a *"bytestring"*. The length of a string can also be the result of an arithmetical operation of other fields and parameters, in fact, in the definition of *"OPCUA_HELLO"*, the *"end_point_url"* is defined based on the message size minus all the other headers except the string.

"*opcua-analyzer.pac*" contains several *"refine"* statements that add callbacks to generate run-time Bro events. The *"refine"* statement is also used to add globally usable function in the context of the OPC UA plug-in. In the same file there is "C++" code that generates the Bro events. Note that the "C++" code is contained between *"%{"* and *"%}"* symbols.

This syntax is necessary in order to instruct the Binpac compiler to behave accordingly. The "*BifEvent*" function is invoked when generating Bro events. The methods are static and follow a common rule: each is named *"generate"*, followed by the name of the event. Before calling them from "C++" they must be formally defined in the *"events.bif"* source file. This kind of sources are simply Bro code for the definition of the events (*"events.bif"*) or of the types used (*"types.bif"*). Note that only a prototype for the types is provided in *"types.bif"*. The comprehensive definition of the OPC UA structures belonging to the plug-in are under *"$BRO_PATH/scripts/base/init-bare.bro"*. This file can be used to add custom environmental definitions (functions, data structures, enumerations, etc..) for the Bro programming.

By observing the various source files of the plug-in, it is possible to note that "*opcua-analyzer.pac*" is the one that contains most of the "C++" code with *"opcua-convert.pac"*. Its purpose is to link OPC UA packets to callbacks that, when fired, will generate Bro events. This file also contains helper functions in "C++" that are used to correctly generate structures to pass to the Bro policy scripts environment. Note that passing the parameters directly in the event generation is not possible; events must be defined in *"events.bif"* before being used in *"opcua-analyzer.pac"*. *"types.bif"* instead define prototypes for the types needed during the event generation. The full definition code for the OPC UA types has been added to *"init-bare.bro"*.

"opcua-convert.pac" contains all the "C++" functions that will be used at runtime by the parser every time it will succeed to parse a valid OPC UA packet to convert its data to valid Bro scripts data structures like *records* and *vectors* of the appropriate type. The code in the source at issue might look boring and repetitive but is how Binpac and Bro scripts are basically interfaced. That code, allocates new structures of Bro types and fill them with the freshly parsed values so that then an event can be generated and the structures juts created can be passed to it.

To define a useful set of API to be used by Bro scripts it is not straightforward. First, events has to be prototyped in the appropriate file. Then, types used in the events has to be prototyped in the types file. Finally, types definitions have to be written comprehensively in the "*init-bare.bro*", so that when Bro loads up it has the types loaded in the "*OPCUA*" module. Furthermore, "opcua-analyzer.pac", includes "opcua-convert.pac", that fundamentally defines all the conversion routines to interface the Binpac parser and the Bro policy script framework. To successfully achieve this task, a special set of API provided by Bro is used. The author succeeded in correctly parsing *integers*, *unsigned integers*, *booleans*, *strings*, *dates* and handled the creation of *structures*, to give complete access of the data to Bro policy scripts.

To give a general overview and understanding of Binpac code, the following listing includes the definition of the "*OPCUA_MESSAGE*" from "opcua-message.pac" file of the plug-in:

```
type MESSAGE() = record{
        type_id: uint8[4] &byteorder=littleendian;
        content : case ( type_id[3] | (type_id[2] << 8) ) of {
                SECURE_CHANNEL_REQUEST -> open_secure_channel_request:
                    OPEN_SECURE_CHANNEL_REQUEST;
                SECURE_CHANNEL_RESPONSE-> open_secure_channel_response:
                    OPEN_SECURE_CHANNEL_RESPONSE;
                PUBLISH_REQUEST -> publish_request:
                        OPCUA_PUBLISH_REQUEST;
                PUBLISH_RESPONSE -> publish_response:
                        OPCUA_PUBLISH_RESPONSE;
                CLOSE_SECURE_CHANNEL -> close_secure_channel: OPCUA_CLOSE_CHANNEL;
                BROWSE_REQUEST -> browse_request:
                    OPCUA_BROWSE_REQUEST;
                BROWSE_RESPONSE -> browse_response:
                    OPCUA_BROWSE_RESPONSE;
                READ_REQUEST -> read_request:
                        OPCUA_READ_REQUEST;
                READ_RESPONSE -> read_response:
                    OPCUA_READ_RESPONSE;
                WRITE_REQUEST -> write_request:
                        OPCUA_WRITE_REQUEST;
                WRITE_RESPONSE -> write_response:
                    OPCUA_WRITE_RESPONSE;
                CREATE_SUBSCR_REQUEST -> subscription_request:
                    OPCUA_CREATE_SUBSCR_REQUEST;
```

```
                CREATE_SUBSCR_RESPONSE -> subscription_response:
                    OPCUA_CREATE_SUBSCR_RESPONSE;
                GET_ENDPOINTS_REQUEST     ->      get_endpoints_request:
                    OPCUA_GET_ENDPOINTS_REQUEST;
                GET_ENDPOINTS_RESPONSE ->         get_endpoints_response:
                    OPCUA_GET_ENDPOINTS_RESPONSE;
                default                                                    ->
                    not_parsed: empty;
        };
}
```

It is possible to observe that the data structure "MESSAGE" is defined as a type, that is basically a sequential collection of fields that have other types. A message might be of several types, in fact the "case" statment identifies the distinction between the different OPC UA types. Note that the "case" construct uses the "type_id" field to determine the type of the "content" field. If, for example, a packet "type_id" matches the "SECURE_CHANNEL_REQUEST" value, then the parsing will proceed by taking "OPCUA_SECURE_CHANNEL_REQUEST" as type of the field and therefore as the next structure to be parsed. The Binpac code in the end consists of formal definitions that formally describe the expected packet structures. For a raw, comprehensive definition of the supported types refer to "opcua-types.pac"

One of the most basic functionalities of IEC 62541 are the *"OpenSecureChannel"* and *"CloseSecureChannel"* services. Citing the sixth document of the standard: "These Services specify how to establish a Secure Channel and how to apply security to Messages exchanged over that Secure Channel" [48]. These two services are fundamental to open secure communications between OPC UA hosts. Following, the content of the "opcua-request.pac" file, that contains the formal definition of the "OpenSecureChannel" request.

```
type OPEN_SECURE_CHANNEL_REQUEST() = record{
        header: REQUEST_HEADER;
        client_protocol_version: uint32;
        security_token_request_type: uint32;
        message_security_mode: uint32;
        client_nonce: STRING;
        request_lifetime: uint32;
}

type REQUEST_HEADER() = record{
        authentication_token: NODE_ID;
        timestamp: uint64;
        request_handle: uint32;
        return_diagnostics: uint32;
        audit_entry_id: uint32;
        timeout_hint: uint32;
        additional_header: uint8[3];
}
```

The previous code also defines the type *"REQUEST_HEADER"*, that is used in almost all the OPC UA requests. However, the main definition of the file is the type of packet that is sent when a client wants to establish a secure channel to a server. It is straightforward to note that the structure at issue is pretty short and it only contains integers, except for the "client_nonce" field, that is a variable length string. Furthermore, the *"OPEN_SECURE_CHANNEL_REQUEST"* is also referred in a successive section, 3.2.5, in which a potential vector for DoS outbreaks in described.

The plug-in is organized so that Binpac source files are of a short length each so that most of them can be visualized in just one screen without the need to scroll. The files are named as the services or messages they implement. Some source just defines enumerations that will be used to distinguish the different OPC UA packets' types, while, other files contain conversion function to pass data from the OPC UA plug-in to the Bro policy script framework.

A critical issue in the parser development is the testing for robustness, because memory corruption flaws nested in the program would set the right conditions for exploitations of the aforesaid vulnerabilities by anybody that wants to take down or control the NIDS. To test robustness, exception logging has been employed, to detect malformed packets that

might be source of an attack, or simply packets not recognized by the parser. To give as much coverage as possible during the testing all the available packets will be given in input to the parser to check its behavior.

Performances are also important because if bad enough they can compromise the robustness of the system (Bro's packets queues might exhaust memory). However, in this project, good performances should be inherited by default from the favorable Binpac's performances, discussed in section 2.3.3. Binpac was implemented to fulfill features such as robustness and performances, but as a user of Binpac, the author had to evaluate the effectiveness of the security and performances of the system. That will be discussed in section 3.3.

3.2.2 Compilation

The process of compilation must first turn Binpac sources into valid "C++" code to then embed it as a plug-in in Bro at compilation time. Bro is designed to have different directories for each plug-in. To initialize a new extension, with its own files and folders, it is necessary to invoke the "binpac_quickstart" tool [2]. After termination, there will be two directories, named as the plug-in, for the sources presented in section 3.1.1 and in the parser directory there will be a *"CMakeLists.txt"* file to configure the compilation process.

There are two compilation steps that must be achieved to have a working Bro plug-in:

- Compile Binpac to "C++" (with Binpac)
- Compile Bro with new plug-in (with GCC)

In the first one there might occur errors generated by the Binpac compiler and strictly related to its own syntax and lexicon. Once the plug-in is compiled, it is necessary to invoke GCC to build Bro and incorporate the new plug-in. The second compilation step is usually more complex because the "C++" code written in the Binpac sources is validated at this time and it is generally more error prone. The "C++" code is more subject to mistakes because it has to deal with Bro API to generate events and feed them with appropriate Bro types. Therefore, is a part of the process where it is easy to generate a compile time type error. The developer needs to get acquainted with the "C++" types and the surrounding API to avoid mistakes.

In the parser's folder there is a file named as the plug-in, that is usually named as the protocol to parse. The aforesaid file, in this plug-in, is *"opcua.pac"* and it is the root of the Binpac's compilation process. It includes the basic files needed: Bro's types and events, the Binpac analyzer and the protocol definition file. The BiF files have ".bif" extension and contains code very similar to the Bro code with a slightly different syntax. "events.bif" defines the events that the plug-in will pass to the Bro policy script interpreter while "types.bif" defines the additional Bro structures introduced to be used as parameters in the events (otherwise, passing non-complex types all the time would result in very long lists of parameters, ruining the code readability and maintainability). "opcua-protocol.pac" contains the formal definition of "*OPCUA_PDU*", from which all the other packet structures will be derived. "opcua-analyzer.pac" contains the needed function primitives that bind callbacks that will be fired on successful parsing of OPC UA messages. Those callbacks will then generate Bro events using Bro's API.

To compile the Binpac sources it must be invoked the Binpac compiler by providing in input the main source file, "opcua.pac". To compile, it is also needed to give it the include directory. The final command is:

$$binpac\ -I\ \$BRO\_PATH/src\ opcua.pac$$

In case of errors, Binpac will try to inform the user about the problem with explanatory messages. Once the above command succeed, the OPC UA Binpac source folder will also contain "C++" code, result of the Binpac compilation.

Now it is time to include the new analyzer in Bro. To do that the compilation process must be executed one more in the "$BRO_SOURCE_PATH", where also the plug-in described in this thesis is nested. To achieve that, a simple command from the source root directory is required:

$$make$$

It is possible to observe that *"cmake"* will cleverly skip all the files that have not been altered, while the freshly modified OPC UA plug-in will get compiled and embedded in the Bro structure. This happens because the plug-in folder contains a "CMakeLists.txt" file and *cmake* is programmed to seek for directives in it. After compiling the plug-in related file, *"cmake"* will link the Bro executable and then it will be possible to install it in the system by typing:

sudo make install

Or just use it from the build, "*$BRO_SOURCE_PATH/build/src/bro*".

To summarize: the user should first compile the Binpac source, then he should compile the generated "C++" sources in the Bro compilation to get the plug-in to run. The final result is Bro executable with OPC UA support. At this point, the user can run Bro with some policy scripts, to make NIDS analysis.

Note that the author also provides a script to compile, link and embed the final work. Furthermore, it is important the the following steps are respected during the compilation and incorporation of the plug-in.

- Compile Binpac source to "C++"
- Compile "C++" into machine code
- Install or link the new Bro executable
- Copy the necessary Bro's init scripts from the developing folder to the system installation path
- Run Bro with custom OPC UA policy scripts in input

The script provided by the author complies with the installation procedure. But it requires the user to modify the bash variables in the first lines of code to define the Bro's source location and installation path. Refer to appendix A for the integral "BASH" script to compile and install the plug-in.

Note that while the Binpac sources are all grouped under the same directory, the Bro policy scripts are spread because of the Bro structure. In the default OPC UA policy directory there are functions that log the events to log files. In "init-bare.bro" there are basic structure definitions that are used to generate the parameters of the API (events) passed to running Bro scripts. Furthermore, user-defined custom Bro programs can also be provided and executed by the final user.

Bro scripts do not need to be compiled to run. As previously stated, there is an open project, from the Bro community that aims at writing a binary compiler for Bro scripts [6]; but for the moment the efficient interpreter is the only working solution to run policy scripts.

3.2.3 Bro policy scripts

Bro policy scripts are the instrument used to implement the core network intrusion detection mechanism. Theoretically, these scripts are arbitrary Turing complete programs written in the Bro programming language. Practically, these will always tend to implement anomaly or signature based detection of specific attacks, or, they might also log transient information.

Upon the OPC UA plug-in written in Binpac, discussed in section 3.2.1, it becomes possible to write Bro policy scripts. They will receive events with structured data from the OPC UA parser. This interface is developer friendly because it simplify the job of writing policy scripts. Without such a mechanism it would be necessary to write the parser from scratch.

To support the definition of significant security policies this work integrates detection examples to identify attacks discovered during the security analysis of the OPC UA server official reference implementation conducted by the federal information office of the German government in 2015 [49].

It is important to note that the security analysis made by the information office only covers the official server and client reference implementations (written in "C") and the relative network stack [49]. Main issues identified by the research were: problems related with inconsistencies between the official documentation and the implementation. That might arise security issues as well as not; and bugs related to programming errors, therefore issues regarding the reference

implementation. Those mistakes in the code are more or less dangerous depending on the context, but it has to be pointed out that DoS attacks, are the most common form of outbreak to perform in this context. Even if an attacker does not know how to perfectly exploit a memory corruption vulnerability to control a crash on the target machine he can still use out-of-bounds read and write to make, i.e., the OPC UA server to read and write areas of memory unmapped in its address space and consequently get killed by the operating system [49]. The reference implementation of the server at issue is particularly delicate because written in "C" and hence subject to memory corruption issues on pointer handling errors.

On a security perspective, outbreaks can target OPC UA servers by sending malformed packets. Basically, if the server is deceived into reading or writing out of the buffer it dynamically allocated that will result in buffer overflows. Those flaws, can be exploited to make DoS attacks or to take over machines. From the security analysis on OPC UA, around 400 crashes were identified and analyzer with *valgrind* during the tests [49]. To detect malicious packets, the author employs an anomaly detection mechanism based on catching exceptions. There might be several cases where a packet is erroneous, it also have to consider the case in which the packet is malformed so that it contains wrong length of for example strings or arrays contained in it. In that case, the Binpac parser will fail to find a valid structure for the packet and it will throw an exception that will be displayed on screen or potentially it could also be conveyed elsewhere.

On a goal perspective, detecting malformed packets as well as those that looks valid but contain an attack payload is important to monitor for outbreaks. Another application of anomaly and intrusion detection, in the context of OPC UA is the chance to compare a reference implementation to the official, public standard. That has also been parto of the security analysis of OPC UA [49].

The network intrusion detection system can be used for different purposes that can be grouped as follows:

- Detect attacks
- Notice deviations respect to the standard
- Data processing

This section discusses about the issue of implementing Bro scripts to achieve the goal of defining security policies. Achieving such a task with the Bro language is easier compared to other languages because it was made for that [61]. Bro is a security policies programming language that other than being Turing complete, it supports constructs typical of every imperative programming language like *"if"*, *"while"*, *"function"*, etc. As discussed is section 2.2.3, the language supports variables, as well as constants and all the other feature of modern programming languages.

Bro scripts are simple and powerful. For example, it is easy to keep track of the state of a connection by associating runtime variables with it. Distinguishing connections is also straightforward because Bro provides a "connection" typed parameter by default in each event. The following snippet of code defines a table that will use addresses as keys and unsigned integers as values.

$$global\ my\_table:\ table[addr]\ of\ count;$$

Then, the aforesaid structure can be used for example to count the number of sent messages by a certain host. It is also possible to change the *"count"* typed variable to some more complex data structure to associate more information with the address.

For example it is possible to define a structure designed to store the hosts with which a certain system exchanged information.

$$global\ my\_connections:\ table[addr]\ of\ set[addr];$$

Then the structure can be updated whenever a new TCP connection is establish, or some other event inform us of a digital communication going on between two hosts.

Bro scripts support the use of regular expressions to quickly check presence of patterns in strings. To clarify syntax and usage it follows an example from [10]:

$$/foo|bar/ == \text{“}foo\text{“}$$

Gives true, while:

$$/\hat{}oob/ \; in \; \text{“}foobar\text{“}$$

Returns false. Another straightforward Bro functionality is the availability of key words to quickly write to specific log files custom structures: For example:

$$Log :: create\_stream(Opcua :: LOG, \; [\$columns = Info, \; \$ev = log\_opcua, \; \$path = \text{“}opcua\text{“}]);$$

Will create a log file (or opens it if already existing) that will be updated by the *log_opcua* event. The path instead specifies the file name, while the first paramenter informs about which structure the log file will have. Therefore, using this simplifications it becomes possible to implement custom and flexible logging strategies quickly.

To write on the create log stream, a simple call is enough:

$$Log :: write(Opcua :: LOG, \; info);$$

All the files in “$BRO_PATH/scripts/base/protocols/opcua/” are the base OPC UA security policy scripts that are automatically loaded during each Bro execution if the “opcua” module is enabled in the configuration file. Those files are useful for setting basic data processing operations like to log targeted information or also to record everything. Potentially it is possible to implement all the policies in the base directory but it is not mandatory in any sense. It is better to develop several modules but with a precise goal, dividing the functionalities by pertinence.

OPC UA is used in ICS and IoT. DoS attacks on the industry are yet rare but very hard to recover from. Stopping production machineries or even destroying them might cost millions of Euro. Consequently, this document will mainly concentrate on detecting DoS attacks identified by [49], while not forgetting to also include examples to detect deviations from the specifications.

In the next sections are discussed some examples of detection techniques implemented during this thesis.

3.2.4 Incorrect sequence numbers

As stated in [49], the OPC UA server tested during the analysis behaved by accepting incorrect sequence numbers. This, should trigger a transport error [48] pag. 39.

The security analysis on OPC UA [49] at section 8.9 proposes therefore a Proof of Concept (PoC) of a replay attack based on the fact that the server accept messages with wrong sequence numbers as a follow-up to the project.

If an attacker exploits this flaw, he might be able to exhaust the server resources by replay attacking the victim server with packets that the attacker knows or suppose they will provoke a high resource consumption in the server and he did eavesdropped in the past. In this way a DoS outbreak can be easily achieved. But an attacker might also be able to bring a connection into fatal errors by sending recorder packets twice or more times, depending on the implementation of the server.

Replay attacks are all based on the concept that if there is no proper mechanism in place, an attacker with eavesdropping capabilities might record the packets sent from the clients, replay them in the future and get them accepted by the server. What is required to prevent replay attacks is some mechanism to make packets uniquely valid or at least, very difficult to reuse for an external wiretapper. Examples of solutions are sequence numbers, that must match the supposed value otherwise the packet gets discarded. Furthermore, sequence numbers have to be enforced with proper encryption and authentication mechanisms as well. Nonces are also useful to generate fresh signatures or Message Authentication Codes (MAC) of the content that must be proved to come from a certain sender [45]. A tuple made by: message,

nonce and sequence number, where the message is arbitrary, the nonce is stochastically unique and the sequence number is periodically repeated will give every time different signatures if signed with a fixed asymmetric private key. If that signature is then actually verified by the server, then replay attacks are basically prevented.

Nonetheless, for the flaw at issue, as a detection mechanism, it is possible to realize a relatively simple policy script that per each connection monitors that the sequence numbers are actually sequential, otherwise it raises an alert. It is possible to achieve that by keeping a table in memory with the current maximum sequence number per each connection. To identify the specific link, it uses the tuple that identifies the TCP connection, that is made up of IP and ports of the connected hosts.

$$global\ max\_sequence\_number : table[conn\_id]\ of\ count;$$

And then it is possible to check when there is a message of type *"MSG"*, if the sequence number is the successor of the received one. The script implement four data structures that are required to keep track of all the necessary information. The first one identifies per each connection, the highest sequence number received, while the other keeps track for each connection, any pair of sequence number, request identifier is saved, so that the script knows which couples of requests and responses have been already sent. This is important because a client may send multiple sequential requests from a single TCP socket to a server, while expecting to get all the valid responses back with same sequence number and request identifier that it associated with them. Refer to appendix B for the whole script.

The script does also detect when requests are sent instead of responses and also the other way around. Note also that the code manages the "HEL" typed message, that is used to establish communications. Indeed, that the event reset the state of sequence numbers, requests and responses associated with the connection. The code listed also keeps track of request response pairs, to give more information during the logging and to detect if wrong type of messages are sent. The routine implemented in the "bro_done" event checks if for each request a response has been received. If the program detects incomplete pairs, it notices that by logging to the standard output.

The script uses two different tables: one to store the current sequence number per each connection and the other to keep in memory a boolean for each connection, that identifies if a certain sequence number has already been seen. This is necessary because in OPC UA, the sequence numbers should match on request response pair; while different requests responses have disparate sequence numbers.

The logic of the script is not too complex and the comments should help the reader in understanding the implemented logic. The script basically keeps track for each connection of the maximum sequence number, in "*max_sequence_number*" and it also tracks each request response pair in "*sequence_numbers*". The logic is the following: if a message is a request, update the relative structure, then, check if the sequence number is valid, if true, update the relative structure, otherwise report the error. If it is a response, update the corresponding structure. If it receives a pair for which request and response have already been sent for that connection, report the error.

Note that the code implements an event that function as an API from the parser as already discussed in section 2.2.2. To get further information on events, see appendix D. To discover the structure of the "params" argument of the event, or of other parameters with custom types, refer to appendix C.

By putting the code in appendix B in a file named with extension ".bro" and giving it as final argument when running Bro with the OPC UA plug-in loaded will analyze all the OPC UA traffic using the provided policy scripts. Note that is also possible to load other scripts in Bro source using the keyword "*load*".

This first example, accomplishes the goal of detecting inconsistencies in the sequence numbers, practically applying a security policy. But, there are other ways to implement the concept that may reduce or increase the log produced by the script based on varying needs. Also note that some scripts might need to be calibrated depending on the network on which they need to run. Scripts that detect DoS attacks in general are more prone to produce false positives and therefore require careful configuration.

Detecting packets with invalid sequence number is a PoC that Bro policy scripts can be used to enforce the security policies by providing smart monitoring of the network flow and therefore the possibility to stand by for attacks, or also, to supervise OPC UA implementations to reveal deviations from the standard.

3.2.5 HEL flooding

Section 7.2 of the security analysis of OPC UA discusses several potential threats to the protocol [49]. All the findings has been classified using several criteria in a Common Vulnerabilities Scoring System (CVSS) style. Factors taken into account were: Access Vector (AV) as well as Access Complexity (AC), Authentication (Au), Confidentiality Impact, Integrity Impact, Availability Impact, Exploitability [49]. Then, based on the likeliness, impact and the other characteristics, each flaw was given a score between 0.7 to 8.5 [49].

The vulnerability at focus on this section was named "HEL flooding" because during the security analysis, they discovered that if the clients repeatedly send "HEL" messages to the server, it will reply with an "ACK" for each one. That behavior is not the correct one as specified in the documentation, therefore, the impact of the vulnerability has been classified as a potential DoS attack because it provokes high network load and low processor load [49]. The vulnerability is present in all the security modes of OPC UA. The final CVSS score given during the security analysis was 5.2 [49].

The server behavior is incorrect because while it should drop the repeating "HEL" messages, it instead normally reply to with an "ACK" per each one, therefore consuming resources and giving space for a DoS breach.

To detect "HEL" flooding with a Bro script it is pretty straightforward. The following listing shows the very short script the author wrote for this purpose.

```
module OPCUA;

#max number of HEL from each IP
const threshold: count = 10;
const timelapse: interval = 10 sec;

global account_hello: table[addr] of count &write_expire=timelapse;

event opcua_hello_event(c: connection, params: connection_parameters, end_point_url:
    string){
  if( c$id$orig_h in account_hello )
    account_hello[c$id$orig_h]+=1;
  else
    account_hello[c$id$orig_h]=1;

  if(account_hello[c$id$orig_h]>threshold)
    print fmt("Flooding from: %s",c$id$orig_h);

}
```

The core idea here is that it is possible to log how many "HEL" messages a certain IP is sending to any server. If more than "threshold" requests are issues during a "timelapse" interval, then Bro will notify the likely flooding.

The constants in the script can be calibrated based on the network in which the monitor has to run in. Furthermore, in the listed code, the logic divides requests by IP so that it takes into account the behavior of different hosts. It could be modified to set a global "HEL" threshold and not only a limit per each client.

3.2.6 Version mismatch

During the security analysis of OPC UA steered by the German government, they discovered several inconsistencies between the standard and the implementation and they also found specific flaws in existing OPC UA software. In section 8.6.3, the ninth point lists a finding that basically consists in the fact that the official OPC UA server implementation did accepted "OPEN_SECURE_CHANNEL_REQUEST" with protocol version different than the one that was sent in the "HELLO" message [49]. This does not correspond to the specifications and hence can be considered a deviation from the normal, standard behavior and therefore, an anomaly. It might imply the malfunctioning of the communication because of protocol incompatibility.

Even if this is not an actual security problem, it is possible to write a simple detection script, based on the API described in the appendix D. To detect versions mismatch the author implemented a Bro script that matches the version from the "HELLO" message to the one that aims at opening the secure connection. This script can be useful to detect the anomaly

at issue. If detected, then the reference implementation can be fixed to prevent the server from accepting mismatching version values. Following, the script at issue:

```
module OPCUA;

global hellos_version: table[conn_id] of count;

event opcua_hello_event(c: connection, params: connection_parameters, end_point_url:
    string){
  hellos_version[c$id] = params$version;
}

event opcua_open_secure_channel_response_event(c: connection, header: response_header,
    server_protocol_version: count, security_token: security_token, server_nonce:
    string){
        if( hellos_version[c$id] != server_protocol_version)
    print fmt("Mismatch between versions in HEL msg and OPEN_SECURE_CHANNEL_REQUEST in
       %s",c$id);
}
```

3.2.7 Memory leak with NodeId decoding

During the security analysis of OPC UA, more than 400 error messages had been caught in *Valgrind* [49]. A DoS vulnerability identified and listed in section 8.7.1 of the security analysis was found in the function *"OpcUa_BinaryDecoder_ReadNodeIdBody()"*. The bug laid on the fact that the code was not handling correctly the freeing of memory associated to the "node_id" typed variable if an error occurred during the decoding, therefore generating memory leaks in RAM [49]. In fact, if a malformed packet with a "node_id" that declares it contains a string of a certain length but actually it does not because it terminates previously, the non-patched server will allocate space on the heap for the string but will not free it during the handling of the parsing error. Repeating this attack would trigger total memory consumption on the vulnerable server, therefore blocking the service.

To detect this outbreak it is necessary to have Bro with the plug-in and patches developed in this project to process the OPC UA malformed packets and print the exceptions. If the message error gets generated by and out-of-bound case, on a "node_id" typed value, then, Bro processed one of those malformed packets.

3.2.8 Malformed packets detection

An important feature of the OPC UA parser, core of this project, is that it manages automatically errors with indexes, bounds and other kind of problems by raising exceptions [64]. The author, created a patch that is automatically applied after having generated the parser by the building script. The patch at issue adds prints of the caught exceptions, so that if there is a problem it can be detected and debugged. Another major advantage of printing exceptions is that it becomes possible to detect malformed packets that the parser failed to parse. A reference implementation of OPC UA client or server that does not successfully handle the exception as the Binpac parser would therefore crash or behave unexpectedly.

Figure 15 shows how Wireshark display malformed packets:

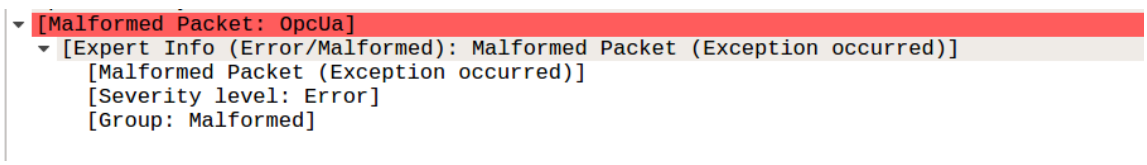

Figure 15: Wireshark OPC UA malformed packet

Figure 16 instead shows how Bro with the plug-in developed in this project with the applied patch detects and inform about the malformed packet.

The Bro exception message is more informative, in fact, it is possible to understand in which type the exception was thrown, for which reason and on which field. In the example, the parser fails during the reading of an *"AR-*

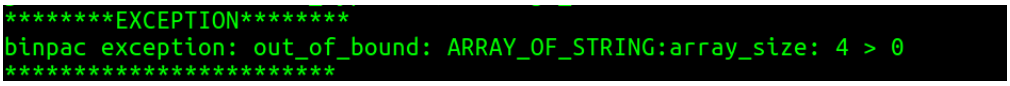

Figure 16: Bro detects OPC UA malformed packet

RAY_OF_STRING", while parsing the "*array_index*" field. The message informs us also on how many bytes the parser expected to read (4) and how many it actually had available on the buffer (0).

OPC UA is used in the ICS and IoT fields. DoS attacks are frightening for the industry because even in the case of medium size farms, the losses for the interruption of the machineries may sum up to huge amounts. An example of ICS malware discovered back in June 2010 was the Stuxnet malware. It was a carefully designed piece of software that infected more than 14 industrial sites in Iran among which an uranium-enrichment plant. It was designed to scan Windows systems for Siemens Step7 software after installation. Then it basically used the Step7 software to program industrial machineries such as centrifuges by compromising the PLC, actually taking over the machine. It was used to spy on the industry as well as to destroy centrifuges by making them spin too fast [37]. In IoT, DoS attacks are not as powerful as in the industry. What happens instead is that IoT devices themselves are exploited to drive flooding based DoS attacks. In fact IoT devices have already been used to create botnets and steer Distributed Denial of Service (DDoS), i.e. in the case of the "Mirai" malware that created one of the biggest botnets ever [56]. The unfortunate lack of security, the quantity of IoT devices and their accessibility through the network makes them a perfect target for cyber criminals.

Detecting malformed packet is useful to understand when attempts of crashing or controlling OPC UA machines are in course. This kind of activity is typical of cyber security employees that have the goal of monitoring and protecting the IT systems. When an exception is caught, a message saying which kind of rejection has been identified is printed. To investigate further of where the malformation lays, it would be ideal to log the whole deformed packet (reassembling chunks if necessary) and further inspect it by adding log information to the Bro source code, or using Wireshark or other tools to understand what is wrong with it. However, the danger of a malformed packet is real only when it brings some reference implementation to crashes or to experience anomalies during the execution.

To test that the parser written during this thesis detect malformed packets, the author generated minimal "pcap" files that have erroneous structure. When testing Bro with the OPC UA plug-in loaded and malformed OPC UA packets in input, it will throw an "out_of_bound", or akin exception and it will print its details on screen. It might be that the malformed packets at issue might trigger vulnerabilities in some OPC UA reference implementation. Bugs in parsing have been repetitively used as vectors to drive memory corruption attacks.

Memory corruption issues are usually correlated with array pointers that exceed the bounds of allocated memory to read or write. In some cases an attacker might be able to decide exactly the locations he wants to override because of a direct control on an unsafe pointer, anyway there should be a level of indirection added by the Address Space Randomized Layout (ASRL) that function as a barrier to prevent memory corruption attacks to succeed in getting arbitrary code execution.

In 2014, one of the worst vulnerabilities identified and disclosed to the public was "Heartbleed", a flaw in the core of the open Secure Socket Layer (SSL) library [17]. The bug was a variable length information disclosure, till 64KB [16], dictated by an out-of-bound read that the server naively performed on certain requests to provide a response. The issue of this memory disclosure was estimated as critical because it exposed the private SSL signing key of the server as well as the other contents of memory such as messages and other keys. Substantially, a remote unauthenticated attacker was able to exploit CVE-2014-0160 to get the private key of the server with which an attacker could impersonate the server therefore threatening each secure communication established to it. Heartbleed was exactly the kind of vulnerability related to out-of-bound reads during parsing, something that the SSL server should have handled well but did not.

Every developer should be aware and avoid those kind of vulnerabilities in the future. Usual parsing errors like bounds checking can escalate into world wide critical bugs that can compromise most of the ongoing secure communications. It should be known, nowadays, especially to computer scientists, IT engineers and programmers in general that computers are weird machines in the sense that sometimes they do not behave as they were expected to. Memory corruption issues are one of the oldest class of programming errors but they do still affect modern computer systems because the problem has not been removed but requires careful programming and intense testing in memory unsafe languages like "C/C++" to be avoided. Furthermore, security tests in form of fuzzing, white, black or gray box testing are good tools to verify that a software behaves as it is supposed to.

3.3 Evaluation

This section evaluates the performances and robustness of the parser developed in section 3.2.1. Furthermore, the detection scripts of sections 3.2.4, 3.2.5, 3.2.6, 3.2.7 are also evaluated in this section.

Different types of tests have been executed to evaluate effectiveness, performances and to stress test the parser and the Bro's detection scripts. All the test have been performed on a desktop machine with 8 gigabyte of RAM, an Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz quad-core Central Processing Unit (CPU) and Ubuntu 16.04.2 LTS 64-bit.

3.3.1 Performances

To test the performances of the parser developed in this thesis, the author ran and measured the execution time of Bro with the OPC UA plug-in loaded, giving in input the available packet capture files (PCAPs). To test the parser, files containing OPC UA recorded packets are necessary. The author used three different PCAP files, one provided by the Fraunhofer SIT institute and the other two were generated by the author using open reference implementations of OPC UA client and server.

Table 1 describes the characteristics of the PCAP files used during the testing. It also includes information regarding the size of the file, the lapse of time in seconds in which the PCAP was recorded and the total OPC UA payload in the file (only the OPC UA data, excluding all the TCP, IP and data link's headers). The total OPC UA payload has been calculated using a modified version of the script in appendix F.

Table 1: PCAP files used in the performance evaluation

File #	File size	Recording lapse	OPC UA PDU total payload
1	40KB	11 seconds	23.3KB
2	4.8MB	84 seconds	6.3KB
3	10.7MB	1086 seconds	559.62KB

The performances results have been split into two disparate tables to simplify the reading of the data. Table 2 shows the performances of Bro parsing and analyzing different files in "normal configuration". That means no change was made to its settings and therefore all the default plug-ins contained in it were loaded. Each line corresponds to exactly one executed test.

Table 2: Performance evaluation in normal configuration

File #	Execution time (seconds)	Bro configuration	Policy script
1	0.560-0.580	Normal configuration	test.bro
1	0.550-0.560	Normal configuration	NONE
2	0.680-0.720	Normal configuration	test.bro
2	0.680-0.720	Normal configuration	NONE
3	2.180-2.200	Normal configuration	test.bro
3	1.950-1.970	Normal configuration	NONE

Table 3 contains the results of the performance test run with Bro in "minimal configuration". The aforesaid setting consists in running Bro with only the OPC UA plug-in loaded and therefore by not loading all the other plug-ins for the rest of the supported application layer protocols. To achieve this configuration it is needed to edit the Bro settings file that can be found under "$BRO_INSTALLATION_PATH/share/bro/base/init-default.bro".

Table 3: Performance evaluation in minimal configuration

File #	Execution time (seconds)	Bro configuration	Policy script
1	0.330-0.350	Minimal configuration	test.bro
1	0.330-0.350	Minimal configuration	NONE
2	0.460-0.470	Minimal configuration	test.bro
2	0.440-0.460	Minimal configuration	NONE
3	1.000-1.030	Minimal configuration	test.bro
3	0.460-0.470	Minimal configuration	NONE

From table 2 and 3 it is possible to observe that the execution time shrinks substantially when Bro is in "minimal configuration". It is also possible to observe that Bro takes roughly only 100ms more when parsing a file 100 times bigger (first couple versus second pair). Although, effectively, the OPC UA payload is diminished, Bro takes more time to terminate the computation, because anyhow it has to process the packets (by skipping all the non OPC UA traffic in minimal mode but still, processing them takes time). Moreover, it is also possible to deduce that Bro takes around 200ms to load up all the plug-ins it has to parse and setup the network intrusion detection system for all the supported protocols because that is the concrete difference in time that there is between the same tests performed in "normal" and "minimal" mode, except for the biggest tested file. This last test does reveal one second of execution time difference between "normal" and "minimal" mode with the base policy script provided. The execution lapses probably differ so much because of all the traffic of other protocols contained in the capture file at issue.

The column policy script indicates if during the tests a Bro program was given in input from the command line. "NONE" means that no script was given. "test.bro" is the default security policy script that the author developed to test the functionalities of the plug-in and it basically logs various information contained in the OPC UA traffic. Please refer to appendix E to read the integral script. It is important to note that the presence of a simple policy script such as in this case implies negligible execution time fluctuations in the first eight results of the table. Instead, when processing the last file, which size is the greatest between the test capture files and also with the biggest amount of OPC UA PDU, the policy script column make a real difference in the execution time. This because the amount of logging produced by the script is, in this case, enormous and therefore the time taken by the logging itself results relevant.

Last but not least, the table also states a recording lapse column, that identifies the interval of time in which the capture file at issue was recorded in. The time lapses are much greater than the time that Bro takes to parse and analyze them. This is good, because if Bro would not cope with the throughput of the network then it would loose its real-time analysis capability and it could consume all the available memory by piling packets. Moreover, in case more performances are required, a more powerful computer can be used, or it is even possible to run Bro on a cluster of servers [5].

The 40KB file was provided by the Fraunhofer SIT institute, while the 4.8MB file was generated by the author using the "Ansi C" official reference implementation at [24] for the server and a client with Graphical User Interface (GUI) at [36]. The last capture file, of the size of 10.7MB was generated by a script wrote by the author to fuzz OPC UA packets and therefore generate new samples. Also the counting of the OPC UA PDU payload was achieved by a script wrote by the author.

3.3.2 Robustness

As already discussed, robustness is a critical feature of each software related with the computer networks or also programs that runs with high privilges. Software that takes input from the network can be exploited by malicious actors to perform attacks which possible goals might be: to break into the system, to crash the program, or to bypass it. This section is dedicated at the testing of the robustness of the developed system.

The parser discussed in section 3.2.1 and 3.1.1 has been developed with Binpac and therefore it should be automatically very robust. To prove empirically that the parser is effectively strong, the author manually crafted some malformed packets to test that the parser is actually able to detect the malformation without crashing. Furthermore, in this document, especially in sections 3.2.8 and 3.2.7 are discussed the capability of detecting malformed packets by the parser and how this feature can be used to detect anomalies and attacks.

Testing Bro with malformed packets effectively showed that the Binpac parser is actually robust. Looking at the "C++" code generated by Binpac from the formal specifications gives an idea of how strong the performed checks are. Out of bounds read and write operations are prevented by repetitive but necessary checks. As discussed in section 2.3, Binpac automatically implements controls to verify that the information contained in the packet do not mislead the parser into doing unauthorized operations (like reading from memory location zero). However, from the robustness testing, it resulted that the API created by the author to interconnect Binpac with Bro were buggy. The discovered issues were

related to functions contained in the file “opcua-convert.pac”. The aforesaid problematic functions were all related to array conversion from the Binpac (“C++” environment) to the Bro policy scripts processor (interpreter setting).

The vulnerabilities were related to the fact that the code was assuming that the data structures passed to the conversion functions were well formed because otherwise, Binpac should have managed the error by throwing an exception and discarding the packet at issue. For example, consider the case in which a browse response OPC UA packet contains an array of browse results that has an invalid array size, what actually happened is that Binpac did not threw any exception but proceeded by passing the structured data to the event generator and then to the API (that is the actual data flow). Manually reviewing the incriminated Binpac code revealed that it did effectively not threw out of bound exceptions in such cases. As a solution the author implemented additional checks in each API function that manages Binpac arrays (that are actually “C++” vectors). The additive controls verify that if a vector capacity (the space Binpac allocated for it, because that was the size it was supposed to be) matches the effective vector size. If the check fails, then the code throws an exception so that it will be caught and visualized on screen. In this context the author at first preferred to implement the checks manually in the API. Later the author also reported the problem and a possible fix to the Binpac community.

Figure 17 contains a sample of code from the plug-in API (“opcua-convert.pac”) that manages the conversion of the parsed browse results to the Bro policy script environment. Note that the code contains the additional checks just discussed to prevent the improper usage of the vector object (out of the red square). Initially, before the patch, the original code was only the one contained in the red square plus the first statement of the function. Furthermore, note that the code at issue was buggy for the fact that the “for” cycle utilizes the *“${array.array_size}”* field to determine the exit condition. That field might not match the real array length in some particular corner cases in which Binpac does not handle properly out of bounds exceptions and just passes the data to the API routine. Finally, the purpose of the additional check is in fact to manage the missed Binpac exception. Furthermore, note that the code is “C++” with the Binpac’s syntax: *“${...}”* as a shortcut to access the fields of the parsed structures.

```
VectorVal *results(BROWSE_RESULT_ARRAY *array){
        VectorVal *vv = new VectorVal(internal_type("OPCUA::browse_result_vector")->
            AsVectorType());
        if((*array->records()).size() != (*array->records()).capacity())
                throw binpac::ExceptionOutOfBound("BROWSE_RESULT_ARRAY",(*array->
                    records()).capacity(),(*array->records()).size());
        else if(${array.array_size}!=(uint32)-1)
                for ( uint i = 0; i < ${array.array_size}; ++i ){
                RecordVal *rv = new RecordVal(BifType::Record::OPCUA::browse_result);
                rv->Assign(0, new Val((*array->records())[i]->status_code(),
                    TYPE_COUNT));
                rv->Assign(1, new Val((*array->records())[i]->continuation_point(),
                    TYPE_COUNT));
                rv->Assign(2, reference_vector((*array->records())[i]->references(),
                    (*array->records())[i]->ref_num()) );

                vv->Assign(vv->Size(), rv);
        }
        return vv;
}
```

Figure 17: Bug in the API

To test the robustness three different tests have been run with manually deformed PCAPs. The files were named “malformed-1.pcapng”, “malformed-2.pcapng”, “malformed-3.pcapng”. Initially, the second and the third files were generating a “segmentation fault” because the “C++” code in the API was sliding on the parsed array till a certain length was reached. The problem laid on the fact that the length was not the real one of the array and therefore the program was reading out of the bounds of the array, unallocated memory locations that brught him to dereferencing an object at memory location zero. After the author deployed the patches to the API the problems were solved. Please refer to appendix G for a detailed Valgrind log of the crash. Figure 18 depicts the output of the three tests run in sequence.

```
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortoli/PCAPs/mypcaps/malformed-1.pcapng
********EXCEPTION********
binpac exception: out_of_bound: ARRAY_OF_STRING:array_size: 4 > 0
**********************
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortoli/PCAPs/mypcaps/malformed-2.pcapng
********EXCEPTION********
binpac exception: out_of_bound: ARRAY_OF_READ: 69 > 4
**********************
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortoli/PCAPs/mypcaps/malformed-3.pcapng
********EXCEPTION********
binpac exception: out_of_bound: BROWSE_RESULT:ref_num: 12 > 4
**********************
bortoli@pc-hiwi-33:~/tbortoli/sw/test$
```

Figure 18: Robustness tests

Note that the first malformed packet capture file has a “GetEndpointsRequest” that has the “LocaleIds” array of string with invalid number of elements. Figure 19 depicts the packet loaded in Wireshark:

```
▾ OpcUa Binary Protocol
    Message Type: MSG
    Chunk Type: F
    Message Size: 96
    SecureChannelId: 920
    Security Token Id: 1
    Security Sequence Number: 52
    Security RequestId: 2
  ▾ OpcUa Service : Encodeable Object
    ▸ TypeId : ExpandedNodeId
    ▾ GetEndpointsRequest
      ▸ RequestHeader: RequestHeader
        EndpointUrl: opc.tcp://172.17.16.80:4840
      ▾ LocaleIds: Array of String
          ArraySize: 17
          [0]: LocaleIds: [OpcUa Empty String]
▸ [Malformed Packet: OpcUa]
```

Figure 19: Wireshark malformed packet 1

The second has a read request packet that has an untrue size for the array of reads to perform. Wireshark’s packet dissection follows:

```
▾ OpcUa Binary Protocol
    Message Type: MSG
    Chunk Type: F
    Message Size: 139
    SecureChannelId: 4
    Security Token Id: 1
    Security Sequence Number: 4
    Security RequestId: 4
  ▾ OpcUa Service : Encodeable Object
    ▸ TypeId : ExpandedNodeId
    ▾ ReadRequest
      ▸ RequestHeader: RequestHeader
        MaxAge: 0
        TimestampsToReturn: Source (0x00000000)
      ▾ NodesToRead: Array of ReadValueId
          ArraySize: 41215
        ▾ Array length 41215 too large to process
          ▾ [Expert Info (Error/Undecoded): Array length 41215 too large to process]
              [Array length 41215 too large to process]
```

Figure 20: Wireshark malformed packet 2

The third and last has corrupted length of the array of browse results. Figure 21 shows how Wireshark shows the malformed packet:

No further crashes had happened, except for the problems related with arrays identified in this section. Therefore, the system demonstrates to have robust execution in parsing OPC UA traffic by handling normal packets as well as the malformed ones.

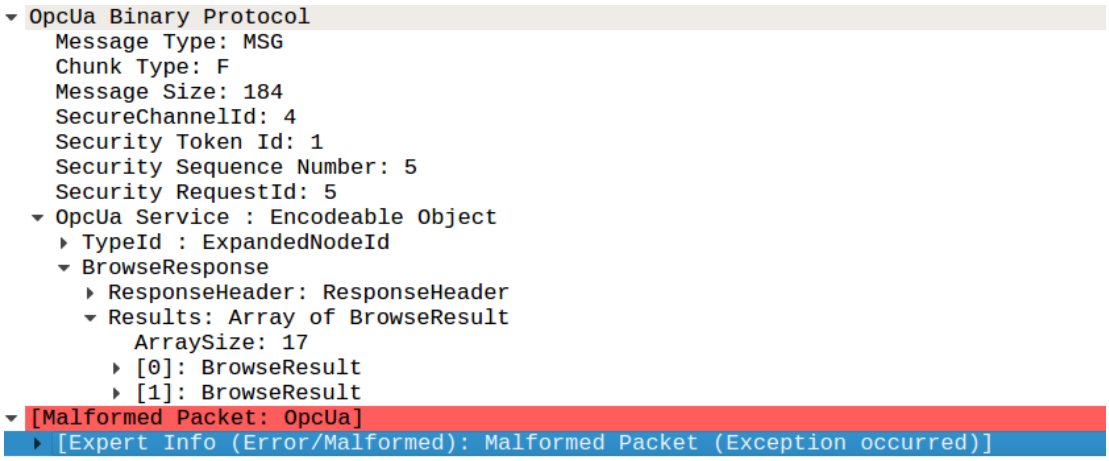

Figure 21: Wireshark malformed packet 3

3.3.3 Sequence number script evaluation

The present master thesis described different security policy scripts that the author wrote to detect disparate kinds of attacks. This section aims at evaluating the effectiveness of those scripts.

The first security policy script, discussed in section 3.2.4, assembles the logic to detect violations regarding the common header of all the OPC UA messages of type “MSG”. Substantially, the checks involve controlling that the sequence numbers of the packets are correct and that for each request there is a corresponding response. The logic can be summarized as follows:

- Each different connection has a disparate, changing state
- Sequence numbers are sequentials, no number can be skipped
- Requests identifiers bind together request and response pairs
- Each request should have exactly one response

Violations to the listed rules produce alert messages from Bro. The printed log can also be conveyed from the standard output of Bro to the email of the responsible administrator, technician, or member of the CERT, so that he is advised as soon as possible. Furthermore, Bro already provide “Notice::ACTION_EMAIL” to send an email to the responsible administrator. However, it might also be desirable to not use emails but some other communication mean, maybe more real-time and secure. It is possible to achieve alluring solutions by reading and filtering the relevant messages from the Bro output.

To test the script the author ran Bro on different instances, to verify that the desired behavior is correct. In this test the author used the policy script at issue in this section as well as the PCAP file provided by the Fraunhofer SIT institute, also used in section 3.3.1, that is a genuine capture file that comes from an actual factory. Running the detection script with this PCAP from appendix B, Bro detects that at the end of the file there are two requests whose have no matching response. That is true and a logical explanation for this fact is that at a certain point the process of recording packets to make the capture file was interrupted and therefore some responses for already logged requests were cut off. The author verified by hand that the requests were effectively with missing response.

Note that each test was run with the “time” command preceding Bro, therefore the outputs contain the time measurements on the execution of Bro. Figure 22 shows the output of the first test:

The script correctly detects that the requests with identifiers 22 and 24 had no responses from the server.

The author performed two more tests to verify that the detection mechanism works properly. Manually deformed PCAPs were used for the tests. The capture file used in this test was modified by tampering the sequence number and the request identifier of one OPC UA packet from 52 to 68 and 2 to 18, respectively. Figure 23 shows the results on giving this capture file, named “wrong_seq_num.pcapng” to Bro.

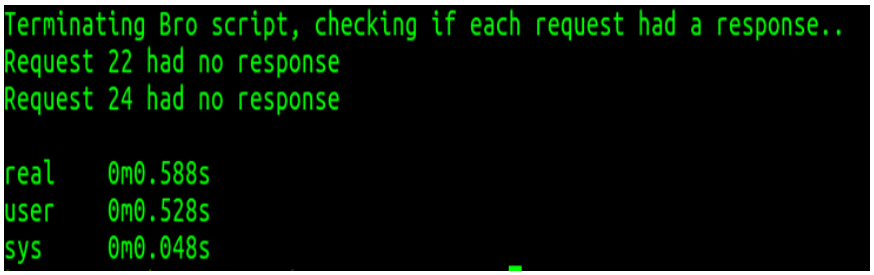

Figure 22: Sequence number detection script - test 1

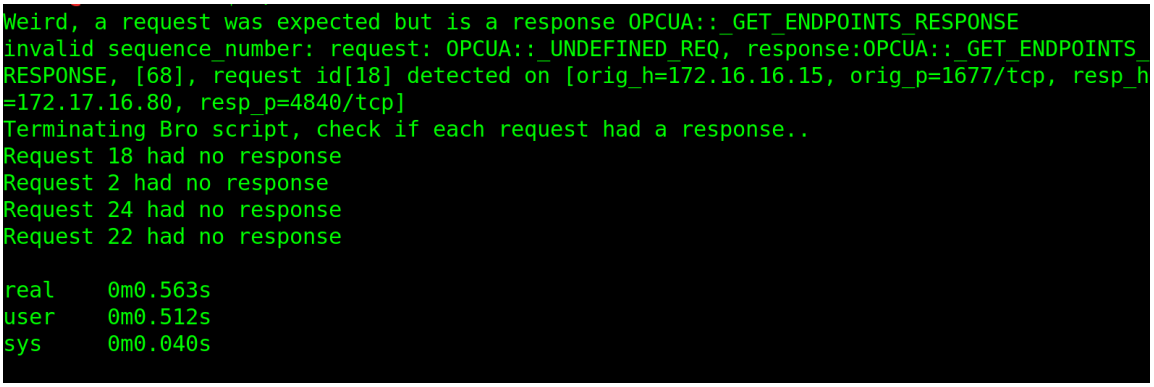

Figure 23: Sequence number detection script - test 2

The script detects that there is a response while a request was expected for that request identifier. It also reveals the altered sequence number and the fact that in the end there are two incomplete request response pairs.

The last test was performed using another manually malformed OPC UA packet, but in this example, the only modified value was the sequence number of an OPC UA packet, changed from 52 to 68. Figure 24 shows the correct output of Bro.

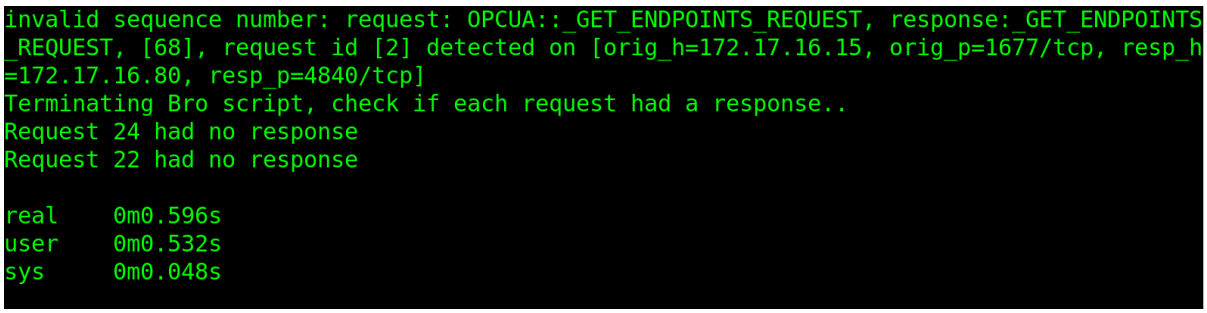

Figure 24: Sequence number detection script - test 3

The script detects the wrong sequence number and the two incomplete request response pairs.

The performed tests show that the developed script is effective in detecting violations of the sequence numbers security policies. As discussed in the section, 3.2.4, the usage of wrong sequence numbers can be exploited by an attacker to forge valid reply attacks because of the flaw identified in [49]. Therefore this policy script is useful to detect and take action to prevent the successful exploitation by malicious actors.

3.3.4 HEL flooding script evaluation

One of the scripts presented in this thesis was the one that detects flooding of "HEL" messages, that is, those communication's rawplug that the clients use to establish and setup an OPC UA connection with a server. The vulnerability, discussed in section 3.2.5, is a non critical but still high rated vulnerability that attackers can exploit to temporarily compromise the responsiveness of a server and therefore all the connected, ongoing communications [49]. Thus, this flaw could be

exploited to threaten OPC UA conversations, on the local network as well as on the Internet. Thence, the author proposed a NIDS script to detect this kind of attack. In this section its effectiveness is evaluated.

To test the security policy script, two different tests were run: one that does not contain any "HEL" flood, while the other with a flood coming from a single host to a server. Both the tests were evaluated positively. To generate the flooding, the author used an OPC UA client with graphical user interface and performed repetitive connections to the server so that the client basically flooded it. Figure 25 portraits the detection script in action on the crafted capture file that actually detects floodings of "HEL" messages from one host.

```
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortol
i/PCAPs/mypcaps/hello_flooding.pcap scripts/hel_flood.bro
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
Flooding from: 141.12.90.143
```

Figure 25: HEL flooding - test

3.3.5 Version mismatching evaluation

Another detection script proposed and implemented in the present document is the one that detects mismatching protocol version between OPC UA clients and servers. The program logic is simple:

- Keep separate state per each connection
- Save in memory the client protocol version when it send an "HEL" message
- Verify that the server protocol version matches when it replies with an "OPN" message

This script proves that Bro policy scripts do not necessarily need to address a security problems but they can also be used as a tool of analysis. This script in fact detects mismatching client server protocol versions. That is not an actual security problem but a violation of the standard that might also lead to malfunctioning in specific conditions [49].

Figure 26 shows the successful test that the author ran the script against a manually modified packet with mismatching client server protocol versions.

```
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortol
i/PCAPs/mypcaps/mismatching_version_numbers.pcapng scripts/version_mismatch.bro
Mismatch between version in HEL msg and OPEN_SECURE_CHANNEL_REQUEST in [orig_h=192.168.0
.133, orig_p=37908/tcp, resp_h=192.168.0.107, resp_p=4840/tcp]
bortoli@pc-hiwi-33:~/tbortoli/sw/test$
```

Figure 26: Version mismatching test

3.3.6 Malformed NodeId detection evaluation

In section 3.2.7, this thesis discussed the memory leak identified during the security analysis of OPC UA [49]. In this section, the detection script implemented to reveal that kind of malformed packets is evaluated. The detection of that kind of attack can be identified by the logging of the exceptions. The following picture shows the execution of Bro with in input a packet with a malformed NodeId element contained in a read request. The aforesaid erroneous structure is a NodeId that specify a string of false length, that is why the exception logs an out of bound in a "'STRING" typed field.

```
bortoli@pc-hiwi-33:~/tbortoli/sw/test$ ~/tbortoli/sw/bro-2.5/build/src/bro -Cr ~/tbortoli/PCAPs/mypcaps/malformed_node_id.pcapng
********EXCEPTION********
binpac exception: out_of_bound: STRING:content: 600 > 15
************************
bortoli@pc-hiwi-33:~/tbortoli/sw/test$
```

Figure 27: Malformed NodeId - test

Figure 28 is the dissection of the packet at issue in Wireshark:

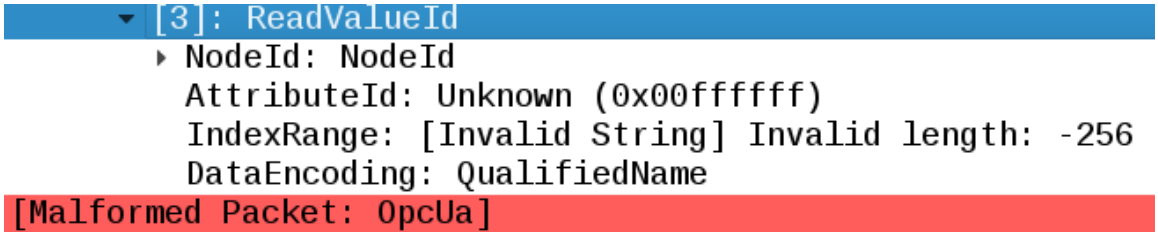

Figure 28: Wireshark malformed NodeId

3.3.7 Summary of the evaluation

Section 3.3 has been allocated to discuss about the evaluation of the contributions committed by author. Section 3.3.1 showed and discussed the results of the performance evaluation. The outcome has been positive, but it has to be pointed out that each network is different and the specific impact of running the system developed in this thesis needs thorough testing and continuous monitoring because in case the NIDS gets overwhelmed by the network traffic, the whole reliability of the system would get compromised.

Section 3.3.2 described the testing steered by the author to assess the robustness of the NIDS. A bugs has been found, reported and fixed by the author regarding the Binpac compiler [1]. No other issues has been found.

Section 3.3.3 discussed the effectiveness of the script developed in section 3.2.4 to detect invalid sequence numbers flowing in OPC UA network streams. The script has been successfully tested against manually modified PCAP files to detect corrupted sequence numbers, request types, and request identifiers.

Section 3.3.4 positively evaluated the 'HEL' flood detection script while section 3.3.5 postitively leaded the testing of the version mismatch detection script. Finally, section 3.3.6 showed how the developed Binpac plug-in is capable of detecting malformed packets crafted to exploit the memory leak vulnerability identified in the BSI analysis [49] discussed in section 3.2.7.

This thesis built a basic framework which purpose is to develop security policy scripts that can be used to process, in real-time, OPC UA data monitored from computer networks and therefore steer analysis of attacks, inconsistencies with the standard, bugs in the implementations and potentially other inventive purposes. This section has evaluated the achieved accomplishments but is not an upper limit to the potentiality of the developed plug-in.

4 Conclusion

Within this thesis the author presented and discussed the relatively new, open, OPC UA standard for M2M communications used in ICS and IoT (sections 1.2, 2.1.2, 2.1). It had then introduced and discussed the topic of NIDS, focusing particularly on Bro and the reasons why it has been the preferred choice (sections 1.3, 2.2). In addition, a distinct section has been dedicated to Binpac, a compiler for application layer network protocol parsers (section 2.3).

This paper then proceeded with the implementation, section 3.2, in which a plug-in for the Bro network intrusion detection system has been developed to add the functionality of parsing OPC UA packets in Bro and therefore making it possible to develop Turing complete security policy scripts in the Bro language to detect attacks, violations of the standard or to implement other arbitrary analysis programs. Specifically, in section 3.1.1, it had been discussed the design of the plug-in, while, section 3.2.1 has been dedicated to the actual implementation of the parser. Then, based on the developed API, the author described and implemented disparate policy scripts in the Bro language to achieve tasks such as detecting attacks revealed in [49], identifying malformed packets and revealing standard violations (sections 3.2.4, 3.2.5, 3.2.6, 3.2.7, 3.2.8).

Lastly, this thesis had discussed the evaluation of performances and robustness of the developed plug-in (sections 3.3.1, 3.3.2), as well as the effectiveness of the developed policy scripts through testing (sections 3.3.3, 3.3.4, 3.3.5, 3.3.6).

In conclusion, Bro demonstrated to be a very flexible and powerful NIDS. Its ability to plug new modules to parse unsupported protocols makes it really ductile and therefore adaptable to present and future network protocols. In addition, Binpac proved to be a very powerful compiler and language to quickly build application layer protocol parsers upon UDP and TCP. Though an issue was identified in Binpac (section 3.3.2), it has to be recognized that it is a great instrument to build application layer network protocol parsers. Besides these observations, OPC UA is a relatively new standard and the only public available security analysis, up to the author's knowledge, is the one from the German government and it has been referenced in the present work as a source of real attacks on OPC UA [49]. Furthermore, no major outbreaks on OPC UA dependent machines had yet been discussed from the public, up to the author's knowledge.

The importance of computer security continues to grow as the world is progressively more interconnected and because of the unbound use of computers which are now involved in more parts of our own lives. In fact, the integration of calculators in IoT, automotive applications and ICS increased recently [60] [21]. Though no relevant attacks on OPC UA happened yet, it does not mean that they will never occur. In fact, the growing importance of IEC 62541 in the industry could imply quite the opposite. Therefore, the plug-in developed in this thesis is valuable as it expands the Bro NIDS on OPC UA. The tool developed in this work can be used as a basis to analyze in real-time OPC UA traffic flows with arbitrarily defined policy scripts toward the purposes of detecting outbreaks, monitoring and safeguarding OPC UA machines.

The main limitation of this work is the coverage of the standard achieved by the implementation. As discussed in 2.1, IEC 62541 is a very wide and flexible standard, composed by 14 documents of formal specifications regarding packets structure, semantic and supposed interpretation by OPC UA clients and servers. The present implementation covers several OPC UA services and data structures but it does not provide comprehensive coverage of the standard. The present work defines a possible solution on how the problem of making NIDS on OPC UA can be practically tackled. It provides concrete solutions to detect discovered attacks [49] on an existing implementation [24]. Furthermore, the solution is also able to detect malformed packets thanks to a fine-grained exception handling mechanism.

The key achievements of this paper are that the developed plug-in effectively unleashes the powerful policy script framework processor on OPC UA traffic as this thesis describes in sections: 3.2.4, 3.2.5, 3.2.6, 3.2.7. More policy scripts can be developed using this basic framework to detect attacks, standard violations, DoS attempts, malformed packets, Advanced Persistent Thread (APT) and other potential abuses. Additionally, the author identified and fixed a programming error in the Binpac software. The author also submitted the problem and solution in the Binpac community so that future crashes will be prevented also for the other users [1].

Future work related to the present paper might target the expansion of the actual API code base of the current plug-in, to support more services and structures of the IEC 62541 standard. Furthermore, in future parsers it will be possible to use "binpac++", the evolution of Binpac, that is unfortunately still under development [6]. With the advantages of "binpac++" the efforts to develop a flexible and robust parser will shrink as more simplifications are being added to save developers' time, especially in the programming of the API for the Bro scripts. With "binpac++" the authors aim at improving the process of developing software parsers by automating more operations. In fact, Binpac requires the developer to write the conversion functions to pass the structured data to the Bro's policy script engine, while,

“binpac++” aims at automating this procedure and therefore the developer will have to manually write less code and thence less errors.

A Compile & install script

This section contains the original script that the author provided to compile and install the developed plug-in. The language utilized is "BASH". Furthermore, the user needs to modify the first lines with its own local paths. Moreover, note that the script includes several checks to assure that Bro and Binpac are actually installed and the script terminates in case of any error (e.g in the case of any compilation error from Binpac or GCC).

```
# Written by Tomas Bortoli <tomasbortoli@gmail.com>
#

# Configure these variables with your own paths
BRO_PATH=~/tbortoli/sw/bro-2.5/
BRO_INSTALLATION_PATH=/usr/local/bro/
SRC=src/ #don't modify
ANALYZER_PATH=src/analyzer/protocol/opcua/ #don't modify

set -e

echo "Checking existence of Binpac installation in /usr/bin/binpac.."
if [[ ! -e /usr/local/bin/Binpac ]]
  then echo "Please install binpac"; exit;
fi
echo "Ok"

echo "Checking existence of bro default installation directory.."
if [[ ! -e /usr/local/bro ]]
  then echo "Please install Bro"; exit;
fi
echo "Ok"
sleep 1

echo "Checking that bro configuration include the OPC UA analyzer.."
check_config=`grep "@load base/protocols/opcua" "$BRO_INSTALLATION_PATH/share/
    bro/base/init-default.bro"`
if [ -z "$check_config" ]
  then echo "Please add a line containing '@load base/protocols/opcua' in
      $BRO_INSTALLATION_PATH/share/bro/base/init-default.bro"; exit
fi
echo "Ok"
sleep 1

echo "Compiling Binpac sources to C++.."
cd $BRO_PATH$ANALYZER_PATH
Binpac -I $BRO_PATH$SRC opcua.pac
echo "Ok"

#patch to print parser exceptions (to catch detect malformed packets)
cd $BRO_PATH
patch -p1 < print_exception.patch

#for safety
mkdir -p $BRO_PATH/build/src/analyzer/protocol/
cp -r $BRO_PATH/src/analyzer/protocol/* $BRO_PATH/build/src/analyzer/protocol/

echo "Recompiling bro to include the updated analyzer.."
sleep 2
cd $BRO_PATH
make
echo "Updating OPC UA  bro scripts, this will need super user's powers.."
```

```
#default policy scripts
sudo mkdir -p $BRO_INSTALLATION_PATH/share/bro/site/base/protocols/opcua/
sudo cp -r $BRO_PATH/scripts/base/protocols/opcua/* $BRO_INSTALLATION_PATH/
    share/bro/site/base/protocols/opcua/

#define OPC UA types structures
sudo cp $BRO_PATH/scripts/base/init-bare.bro $BRO_INSTALLATION_PATH/share/bro/
    base/init-bare.bro

#fix #2
sudo mkdir -p /usr/local/share/bro/base/
sudo cp $BRO_PATH/scripts/base/init-bare.bro /usr/local/share/bro/base/init-
    bare.bro

#update other scripts
sudo cp $BRO_PATH/build/scripts/base/bif/plugins/* $BRO_INSTALLATION_PATH/share
    /bro/base/bif/plugins

echo "Done."
```

B Invalid sequence number detection script

In this appendix section is listed the code of the Bro policy script that detects wrong sequence numbers, mismatching request response pairs and requests without a response.

```
module OPCUA;
type requests_types: enum { _SECURE_CHANNEL_REQUEST, _PUBLISH_REQUEST, _BROWSE_REQUEST,
     _READ_REQUEST, _WRITE_REQUEST, _CREATE_SUBSCR_REQUEST, _GET_ENDPOINTS_REQUEST,
    _CREATE_MONITORED_ITEMS_REQUEST, _CREATE_SESSION_REQUEST, _ACTIVATE_SESSION_REQUEST
    , _TRANSLATE_BROWSE_PATH_TO_NODE_IDS_REQUEST, _UNDEFINED_REQ};

type responses_types: enum { _SECURE_CHANNEL_RESPONSE, _PUBLISH_RESPONSE,
    _BROWSE_RESPONSE, _READ_RESPONSE, _WRITE_RESPONSE, _CREATE_SUBSCR_RESPONSE,
    _GET_ENDPOINTS_RESPONSE, _CREATE_MONITORED_ITEMS_RESPONSE, _CREATE_SESSION_RESPONSE
    , _ACTIVATE_SESSION_RESPONSE, _TRANSLATE_BROWSE_PATH_TO_NODE_IDS_RESPONSE,
    _UNDEFINED_RES};


#keeps max sequence number received per each connection
global max_sequence_number: table[conn_id] of count;
#for each request_id, keeps track if request has been received (F) or req and resp (T)
global request_numbers: table[conn_id] of table[count] of bool;
#tracks requests types
global requests: table[conn_id] of table[count] of requests_types;
#tracks responses types
global responses: table[conn_id] of table[count] of responses_types;


function get_request_type(req: count): requests_types{
        local val: requests_types=_UNDEFINED_REQ;
        if(req==0xbe01)
                val=_SECURE_CHANNEL_REQUEST;
        else if(req==0x3a03)
                val=_PUBLISH_REQUEST;
        else if(req==0x0f02)
                val=_BROWSE_REQUEST;
        else if(req==0x7702)
```

```
                val=_READ_REQUEST;
        else if(req==0xa102)
                val=_WRITE_REQUEST;
        else if(req==0x1303)
                val=_CREATE_SUBSCR_REQUEST;
        else if(req==0xac01)
                val=_GET_ENDPOINTS_REQUEST;
        else if(req==0xef02)
                val=_CREATE_MONITORED_ITEMS_REQUEST;
        else if(req==0xcd01)
                val=_CREATE_SESSION_REQUEST;
        else if(req==0xd301)
                val=_ACTIVATE_SESSION_REQUEST;
        else if(req==0x2a02)
                val=_TRANSLATE_BROWSE_PATH_TO_NODE_IDS_REQUEST;
        return val;
}

function get_response_type(req: count): responses_types{
        local val: responses_types=_UNDEFINED_RES;
        if(req==0xc101)
                val=_SECURE_CHANNEL_RESPONSE;
        else if(req==0x3d03)
                val=_PUBLISH_RESPONSE;
        else if(req==0x1202)
                val=_BROWSE_RESPONSE;
        else if(req==0x7a02)
                val=_READ_RESPONSE;
        else if(req==0xa402)
                val=_WRITE_RESPONSE;
        else if(req==0x1603)
                val=_CREATE_SUBSCR_RESPONSE;
        else if(req==0xaf01)
                val=_GET_ENDPOINTS_RESPONSE;
        else if(req==0xf202)
                val=_CREATE_MONITORED_ITEMS_RESPONSE;
        else if(req==0xd001)
                val=_CREATE_SESSION_RESPONSE;
        else if(req==0xd601)
                val=_ACTIVATE_SESSION_RESPONSE;
        else if(req==0x2d02)
                val=_TRANSLATE_BROWSE_PATH_TO_NODE_IDS_RESPONSE;
        return val;
}

# HEL message resets state of sequence numbers
event opcua_hello_event(c: connection, params: connection_parameters, end_point_url:
    string){
        delete max_sequence_number[c$id];
        delete request_numbers[c$id];
}

event bro_done(){
        print "Terminating Bro script, checking if each request had a response..";
        for(k in request_numbers){
                for(kk in request_numbers[k])
                        if(request_numbers[k][kk]==F)
                                print fmt("Request %d had no response",kk);
```

```
        }
}

# check each MSG message to verify validity of sequence_numbers
event opcua_message_event(c: connection, params: OPCUA::security_params, type_id: count
    ){
        #if first message for this connection, init record in table
        if( c$id !in request_numbers )
                request_numbers[c$id]=table();
  #if trequest_id has never been received for this connection, this is a request
        if( [params$security_request_id] !in request_numbers[c$id] ){
                request_numbers[c$id][params$security_request_id] = F;
                local _t: requests_types=get_request_type(type_id);
                if(_t!=_UNDEFINED_REQ){
                        if(c$id !in requests)
                                requests[c$id]=table();
                        requests[c$id][params$security_request_id] = _t;
                }
                else{
                        local _tt: responses_types=get_response_type(type_id);
                        if(c$id !in responses)
                                responses[c$id]=table();
                        responses[c$id][params$security_request_id] = _tt;
                        print fmt("Weird, a request was expected but is a response %s",
                            _tt);

                }
                #if the request has no valid sequence_number, error
                if( c$id in max_sequence_number && params$security_sequence_number !=
                    max_sequence_number[c$id]+1 ){
                        print fmt("invalid sequence_number: request: %s, response:%s,
                            [%d], request id [%d] detected on %s", (
                            params$security_request_id in requests[c$id]) ? requests[
                            c$id][params$security_request_id]: _UNDEFINED_REQ,(
                            params$security_request_id in responses[c$id])? responses[
                            c$id][params$security_request_id]: _UNDEFINED_RES,
                            params$security_sequence_number, params$security_request_id
                            , c$id);
                }
                #otherwise, update the max_sequence_number for this connection
                else if( (!(c$id in max_sequence_number)) ||
                    params$security_sequence_number > max_sequence_number[c$id] )
                        max_sequence_number[c$id] = params$security_sequence_number;
                #print fmt("%d %d max: %d",params$security_request_id,
                    params$security_sequence_number,max_sequence_number[c$id]);
        }
        #otherwise, check if a response has been already received, if yes throw error
        else if( request_numbers[c$id][params$security_request_id] == T )
                print fmt("invalid sequence_number: request: %s, response:%s, [%d],
                    request id [%d] detected on %s", (params$security_request_id in
                    requests[c$id]) ? requests[c$id][params$security_request_id]:
                    _UNDEFINED_REQ,(params$security_request_id in responses[c$id])?
                    responses[c$id][params$security_request_id]: _UNDEFINED_RES,
                    params$security_sequence_number, params$security_request_id, c$id);
        #otherwise, this is a [potentially] valid response
        else{
                #print fmt("%d %d max: %d",params$security_request_id,
                    params$security_sequence_number,max_sequence_number[c$id]);
```

```
                request_numbers[c$id][params$security_request_id] = T;
                #update tracking of request/response pairs
                local t: responses_types=get_response_type(type_id);
                if(t!=_UNDEFINED_RES){
                        if(c$id !in responses)
                                responses[c$id]=table();
                        responses[c$id][params$security_request_id] = t;
                }
                else{
                        local tt: requests_types=get_request_type(type_id);
                        if(c$id !in requests)
                                requests[c$id]=table();
                        requests[c$id][params$security_request_id] = tt;
                        print fmt("Weird, a request was expected but is a response %s",
                            tt);
                }
                #responses should bring a sequence_number that is smaller than the
                    maximum sequence_number transmitted
                if( c$id in max_sequence_number && params$security_sequence_number >
                    max_sequence_number[c$id] ){
                        print fmt("invalid sequence_number: request: %s, response:%s,
                            [%d], request id [%d] detected on %s", (
                            params$security_request_id in requests[c$id]) ? requests[
                            c$id][params$security_request_id]: _UNDEFINED_REQ,(
                            params$security_request_id in responses[c$id])? responses[
                            c$id][params$security_request_id]: _UNDEFINED_RES,
                            params$security_sequence_number, params$security_request_id
                            , c$id);
                }

        }
}
```

C Bro OPC UA data structures

This section lists the code that define the data structures that the parser generated by Binpac will fill and pass to the Bro policy processor for further analysis.

The following code comes from the “$BRO_PATH/scripts/base/init-bare.bro”.

```
# OPC UA structures definitions
# Written by Tomas Bortoli
module OPCUA;

export{
        type OPCUA::connection_parameters: record{
                version: counter;
                receive_buffer_size: counter;
                send_buffer_size: counter;
                max_msg_size: counter;
                max_chunk_count: counter;
        };
        type OPCUA::node_id: record{
                encoding_mask: counter;
                token: any; #might be: counter, numeric
        };
        type OPCUA::request_header: record{
                authentication_token: OPCUA::node_id;
```

```
        timestamp: time;
        request_handle: counter;
        return_diagnostics: counter;
        audit_entry_id: counter;
        timeout_hint: counter;
        additional_header: counter;
};


type OPCUA::response_header: record{
        timestamp: time;
        request_handle: counter;
        service_result: counter;
        service_diagnostic_flag: counter;
        string_table: counter;
        additional_header: counter;
};
type OPCUA::view: record{
        node_id: counter;
        timestamp: time;
        view_version: counter;
};
type OPCUA::browse_description: record{
        node_id: node_id;
        browse_direction: counter;
        reference_type_id: node_id;
        include_subtype: counter;
        node_class_mask: counter;
        result_mask: counter;
};
type OPCUA::browse_description_vector: vector of OPCUA::browse_description;

type OPCUA::qualified_name: record{
        id: counter;
        name: string;
};
type OPCUA::localized_text: record{
        encoding_mask: counter;
        locale: string;
        text: string;
};
type OPCUA::reference: record{
        reference_type_id: node_id;
        is_forward: bool;
        node_id: counter;
        browse_name: OPCUA::qualified_name;
        display_name: OPCUA::localized_text;
        node_class: counter;
        type_definition: counter;
};

type OPCUA::reference_vector: vector of OPCUA::reference;

type OPCUA::browse_result: record{
        status_code: counter;
        continuation_point: counter;
        references: OPCUA::reference_vector;
};
```

```
type OPCUA::browse_result_vector: vector of OPCUA::browse_result;
type OPCUA::security_token: record{
        channel_id: counter;
        token_id: counter;
        created_at: time;
        revised_lifetime: counter;
};

type OPCUA::notification: record{
        seuqence_number: counter;
        timestamp: time;
        notification_data: counter &optional;
};

type OPCUA::data_value: record{
        encoding_mask: counter;
        variant_type: counter;
        value: any;     #might be: numeric, counter, node_id, qualified_name,
            array_of_string
        status_code: counter;
};
type OPCUA::read: record{
        node_id: OPCUA::node_id;
        attribute_id: counter;
        index_range: counter;
        data_encoding: OPCUA::qualified_name;
};

type OPCUA::data_value_vector: vector of OPCUA::data_value;
type OPCUA::read_vector: vector of OPCUA::read;

type OPCUA::string_vector: vector of string;

type OPCUA::numeric: record{
        namespace_index: counter;
        identified_numeric: counter;
};

type OPCUA::security_params: record{
        secure_channel_id: counter;
        security_token_id: counter;
        security_sequence_number: counter;
        security_request_id: counter;
};

type OPCUA::application_description: record{
        application_uri: string;
        product_uri: string;
        application_name: OPCUA::localized_text;
        application_type: count;
        gateway_server_uri: string;
        discovery_profile_uri: string;
        discovery_url: OPCUA::string_vector;
};

type OPCUA::user_token_policy: record{
        policy_id: string;
```

```
        user_token_type: count;
        issued_token_type: string;
        issuer_endpoint_url: string;
        security_policy_uri: string;
    };
    type OPCUA::user_token_policy_vector: vector of OPCUA::user_token_policy;

    type OPCUA::endpoint_description: record{
        endpoint_url: string;
        server: OPCUA::application_description;
        server_certificate: string;
        message_security_mode: count;
        security_policy_uri: string;
        user_identity_tokens: OPCUA::user_token_policy_vector;
        transport_profile_uri: string;
        security_level: count;
    };
    type OPCUA::endpoint_description_vector: vector of OPCUA::endpoint_description;


}
```

D Bro OPC UA Application Programming Interface

This appendix contains the API definition that the OPC UA plug-in, contribute of the project, provides to Bro policy scripts for analyzing OPC UA traffic. Please refer to appendix C for details on the custom types used in the API.

The following code comes from "$BRO_PATH/src/analysis/protocol/opcua/events.bif" and defines the interface between parser and policy scripts.

```
    # Written by Tomas Bortoli <tomasbortoli@gmail.com>
    #

    # This file defines the events that the analysis generates

    ## c: The connection
    ##
    event opcua_event%(c: connection, chunk_type: count, message_size: count%);

    ## OPC UA Hello event
    event opcua_hello_event%(c: connection, params: OPCUA::connection_parameters,
        end_point_url: string%);
    ## ACK event
    event opcua_ack_event%(c: connection, params: OPCUA::connection_parameters%);

    ## Open/close events
    event opcua_open_event%(c: connection, secure_channel_id: count,
        security_policy_uri: string, sender_certificate: count,
        receiver_certificate_thumbprint: count, sequence_number: count, request_id:
         count%);

    event opcua_close_event%(c: connection%);

    event opcua_message_event%(c: connection, params: OPCUA::security_params,
        type_id: count%);

    ## Open secure channel events
```

```
event opcua_open_secure_channel_request_event%(c: connection, header: OPCUA::
    request_header, client_protocol_version: count, security_token_request_type
    : count, message_security_mode: count, client_nonce: string%);

event opcua_open_secure_channel_response_event%(c: connection, header: OPCUA::
    response_header, server_protocol_version: count, security_token: OPCUA::
    security_token, server_nonce: string%);


## Browse events
event opcua_browse_request%(c: connection, header: OPCUA::request_header, view_
    : OPCUA::view, nodes: OPCUA::browse_description_vector,
    request_max_references_per_node: count%);

event opcua_browse_response%(c: connection, header: OPCUA::response_header,
    nodes: OPCUA::browse_result_vector, diagnostics_info: count%);

# Read events
event opcua_read_request%(c: connection, header: OPCUA::request_header, max_age
    : count, timestamp_to_return: count, nodes_to_read: OPCUA::read_vector %);
event opcua_read_response%(c: connection ,header: OPCUA::response_header,
    results: OPCUA::data_value_vector%);

# Write events
event opcua_write_request%(c: connection%);
event opcua_write_response%(c: connection %);

## Publish events
event opcua_publish_request%(c: connection, header: OPCUA::request_header %);
event opcua_publish_response%(c: connection, header: OPCUA::response_header,
    subscription_id: count, more_notifications: bool, notification: OPCUA::
    notification%);

# Subscription events
event opcua_create_subscr_request%(c: connection, header: OPCUA::request_header
    , publishing_interval: count, lifetime_count: count, max_keep_alive_count:
    count, max_notifications_per_publish: count, publishing_enable: bool,
    priority: count%);
event opcua_create_subscr_response%(c: connection, header: OPCUA::
    response_header, subscription_id: count, revised_publishing_interval: count
    , revised_lifetime_count: count, revised_max_keep_alive_count: count%);

# Get endpoints events
event opcua_get_endpoints_request%(c: connection, header: OPCUA::request_header
    , endpoint_url: string, locale_ids: OPCUA::string_vector, profile_uris:
    OPCUA::string_vector%);
event opcua_get_endpoints_response%(c: connection, header: OPCUA::
    response_header, endpoints: OPCUA::endpoint_description_vector%);
```

E Bro test script

This appendix section lists the code that the author developed to provide a simple policy script to test Bro functionalities, to evaluate performances and to perform logging operations.

```
# Written by Tomas Bortoli <tomasbortoli@gmail.com>
#
module OPCUA;

event bro_init(){
        print "OPC UA policy script init";
        print "";
}

event opcua_event(c: connection, chunk_type: count, message_size: count)
{
        print "";
        print fmt("got OPC UA PDU - chunk_type: %d, message_size: %d",chunk_type,
            message_size);
}

event opcua_hello_event(c: connection, params: connection_parameters, end_point_url:
    string){
        print fmt("got OPC UA hello, version: %d, receive_buffer_size: %d,
            send_buffer_size: %d, max_msg_size: %d, max_chunk_count: %d, end_point_url:
             %s ",params$version,params$receive_buffer_size,params$send_buffer_size,
            params$max_msg_size,params$max_chunk_count,end_point_url);
}

event opcua_ack_event(c: connection, params: connection_parameters){
        print fmt("got OPC UA ack, version: %d, receive_buffer_size: %d,
            send_buffer_size: %d, max_msg_size: %d, max_chunk_count: %d ",
            params$version,params$receive_buffer_size,params$send_buffer_size,
            params$max_msg_size,params$max_chunk_count);
}


event opcua_open_event(c: connection, secure_channel_id: count, security_policy_uri:
    string, sender_certificate: count, receiver_certificate_thumbprint: count,
    sequence_number: count, request_id: count){
        print fmt("got OPC UA open, secure_channel_id: %d, security_policy_uri: %s,
            sender_certificate: %d, receiver_certificate_thumbprint: %d,
            sequence_number: %d, request_id: %d ",secure_channel_id,security_policy_uri
            ,sender_certificate,receiver_certificate_thumbprint,sequence_number,
            request_id);
}

event opcua_close_event(c: connection){
        print "got an OPC UA Close";
}

event opcua_message_event(c: connection, params: OPCUA::security_params, type_id: count
    ){
        print fmt("Got OPC UA MESSAGE - type_id: %x, secure_channel_id: %d,
            security_token_id: %d, security_sequence_number: %d, security_request_id: %
            d",type_id,params$secure_channel_id, params$security_token_id,
            params$security_sequence_number, params$security_request_id);
}
```

```
event opcua_open_secure_channel_request_event(c: connection, header: request_header,
    client_protocol_version: count, security_token_request_type: count,
    message_security_mode: count, client_nonce: string){
        print fmt("open secure channel request, client_protocol_version: %d,
            security_token_request_type: %d, message_security_mode: %d, client_nonce: %
            s ——— %s", client_protocol_version, security_token_request_type,
            message_security_mode, client_nonce, strftime("%D %R", header$timestamp));
}

event opcua_open_secure_channel_response_event(c: connection, header: response_header,
    server_protocol_version: count, st: security_token, server_nonce: string){
        print fmt("secure channel open response — server_nonce: %s — timestamp: %s",
            server_nonce,strftime("%D %R", header$timestamp));
}

event opcua_browse_request(c: connection, header: request_header, view: view, nodes:
    browse_description_vector, request_max_references_per_node: count){
        print fmt("browse request of %d elements — request_max_references_per_node: %d
            ", |nodes|, request_max_references_per_node);
        local i=0;
        while(i<|nodes|){
                print fmt("node_id: %d, reference_type_id: %d, include_subtype: %d,
                    node_class_mask: %d, result_mask: %d ",nodes[i]
                    $node_id$encoding_mask,nodes[i]$reference_type_id$encoding_mask,
                    nodes[i]$include_subtype,nodes[i]$node_class_mask,nodes[i]
                    $result_mask);
                i+=1;
        }
}

event opcua_browse_response(c: connection, header: response_header, results:
    browse_result_vector, diagnostics_info: count){
        print fmt("browse response of %d elements",|results|);
        local i=0;
        while(i<|results|){
                print fmt("status_code: %d, continuation_point: %d, references: %d",
                    results[i]$status_code, results[i]$continuation_point, |results[i]
                    $references|);
                local j=0;
                while(j<|results[i]$references|){
                        print fmt("reference_type_id: %d, is_forward %d",results[i]
                            $references[j]$reference_type_id$encoding_mask, results[i]
                            $references[j]$is_forward);
                        j+=1;
                }
                i+=1;
        }
}

event opcua_read_request(c: connection, header: request_header, max_age: count,
    timestamp_to_return: count, nodes_to_read: OPCUA::read_vector){
        print fmt("read request of %d elements, max_age: %d, timestamp_to_return: %d",
            |nodes_to_read|, max_age, timestamp_to_return);
}
event opcua_read_response(c: connection, header: response_header, results:
    data_value_vector){
        print fmt("read response of %d elements",|results|);
        local i=0;
```

```
        while(i<|results|){
                print fmt("encoding_mask: %d, variant_type: %d",results[i]
                    $encoding_mask, results[i]$variant_type);
                i+=1;
        }
}

event opcua_write_request(c: connection){
        print "write request";
}
event opcua_write_response(c: connection){
        print "write response";
}

event opcua_publish_request(c: connection, header: request_header){
        print "publish request";
}
event opcua_publish_response(c: connection, header: response_header, subscription_id:
    count, more_notifications: bool, notification_: notification){
        print fmt("publish response - subscription_id: %d",subscription_id);
}

event opcua_create_subscr_request(c: connection, header: OPCUA::request_header,
    publishing_interval: count, lifetime_count: count, max_keep_alive_count: count,
    max_notifications_per_publish: count, publishing_enable: bool, priority: count){
        print "subscription request";
}
event opcua_create_subscr_response(c: connection, header: OPCUA::response_header,
    subscription_id: count, revised_publishing_interval: count, revised_lifetime_count:
     count, revised_max_keep_alive_count: count){
        print fmt("subscription response, revised_lifetime_count: %d ",
            revised_lifetime_count);
}

event opcua_get_endpoints_request(c: connection, header: OPCUA::request_header,
    endpoint_url: string, locale_ids: OPCUA::string_vector, profile_uris: OPCUA::
    string_vector){
        print fmt("got endpoints request - endpoint_url: %s",endpoint_url);
        local i=0;
        while(i<|locale_ids|){
                print fmt("[%d] = %s", i, locale_ids[i]);
                i+=1;
        }
}
event opcua_get_endpoints_response(c: connection, header: OPCUA::response_header,
    endpoints: OPCUA::endpoint_description_vector){
        print "got endpoints response";
}
```

F Fuzzing script

This section is dedicated to the script that the author wrote to generate and send mutated OPC UA packets to an OPC UA server, with the purpose of generating malformed packets to test the robustness of the Binpac's parser. The program listed in this section does not have to be considered a finalized working program but a Python script written from scratch, that has been further modified to count the OPC UA effective data in PCAP files and to fuzz similarly to as it is done here.

```
from scapy.all import *
import copy
```

```
import socket
from random import *
from datetime import datetime
from time import *

seed(datetime.now())

global s, s_ip, counter
# a - array to seek in
# b - array to find
def findSubArray(a, b):
    i = 0
    j = 0
    found = False
    if len(a)<len(b):
        return -1

    while i<len(a):
        while i+j<len(a) and j<len(b) and a[i+j]==b[j]:
            j+=1
        if len(b)==j:
            found=True
            break
        i+=1

    if not found:
        return -1
    else:
        return i

def mutateRandByte(array):
    global counter
    for i in range(2):
        array[randint(0,len(array)-1)]=randint(0,255)
    print "mutated"
    print array
    return array
def establish():
    global s
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1",4840))
def sendTest(payload):
    global s
    p = Raw(load=payload)
    s.send(bytes(p))

counter = 0
s_ip=''
establish()
pkts = PcapReader('./tbortoli/PCAPs/mypcaps/test.pcapng')
for p in pkts:
    counter+=1
    print counter
    print p['IP'].src
    #first packet is assumed to have the IP of the host we emulate
    if s_ip=='':
        s_ip=p['IP'].src
    if s_ip!=p['IP'].src:
```

```
        continue

    b = bytearray()
    b.extend(str(p))
    print "original"
    print b
    c = bytearray()
    c.extend(str(p['TCP']))
    print "tcp"
    print c
    print "app"
    pp = copy.deepcopy(p) # deep (recursive) copy
    pp['TCP'].remove\_payload()
    print b[findSubArray(b,c)+len(pp['TCP']):]

    #if contains PAYLOAD upon the TCP header, mutate and send
    if len(b[findSubArray(b,c)+len(pp['TCP']):])>0:
        sendTest(b[findSubArray(b,c)+len(pp['TCP']):])
```

G Valgrind log

This appendix section has been allocated to be a direct reference to the Valgrind log of the execution of Bro that brought to the crash of the plug-in discovered and fixed during the robustness tests, section 3.3.2 [39]. This crash log helped the author in identifying the problem that arose because of an out-of-bound exception that Binpac was missing (Binpac's bug). Later reported and fixed by the author [1].

```
==14061== Memcheck, a memory error detector
==14061== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14061== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==14061== Command: /home/bortoli/tbortoli/sw/bro-2.5/build/src/bro -Cr /home/bortoli/
    tbortoli/PCAPs/mypcaps/malformed-2.pcapng
==14061==
==14061== Conditional jump or move depends on uninitialised value(s)
==14061==    at 0xBC2204: binpac::OPCUA::read_vector(binpac::OPCUA::ARRAY_OF_READ*) (
    opcua_pac.cc:5209)
==14061==    by 0xBBEFA0: binpac::OPCUA::OPCUA_Flow::proc_opcua_read_request(binpac::
    OPCUA::OPCUA_READ_REQUEST*) (opcua_pac.cc:4867)
==14061==    by 0xBB879B: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
    *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2660)
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
    char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
    unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
    binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
    unsigned char const*) (opcua_pac.cc:4726)
==14061==    by 0xBB1FB7: binpac::OPCUA::OPCUA_Conn::NewData(bool, unsigned char const
    *, unsigned char const*) (opcua_pac.cc:48)
==14061==    by 0xBAFBDC: analyzer::opcua::OPCUA_Analyzer::DeliverStream(int, unsigned
    char const*, bool) (OPCUA.cc:60)
==14061==    by 0xD4D0D8: analyzer::Analyzer::NextStream(int, unsigned char const*,
    bool) (Analyzer.cc:245)
==14061==    by 0xD4D542: analyzer::Analyzer::ForwardStream(int, unsigned char const*,
    bool) (Analyzer.cc:331)
==14061==    by 0xCA0237: analyzer::tcp::TCP_Reassembler::Deliver(unsigned long, int,
    unsigned char const*) (TCP_Reassembler.cc:455)
==14061==
```

```
==14061== Invalid read of size 8
==14061==    at 0xBC21F9: binpac::OPCUA::read_vector(binpac::OPCUA::ARRAY_OF_READ*) (
   opcua_pac.cc:5209)
==14061==    by 0xBBEFA0: binpac::OPCUA::OPCUA_Flow::proc_opcua_read_request(binpac::
   OPCUA::OPCUA_READ_REQUEST*) (opcua_pac.cc:4867)
==14061==    by 0xBB879B: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
   *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2660)
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
   char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
   unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
   binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
   unsigned char const*) (opcua_pac.cc:4726)
==14061==    by 0xBB1FB7: binpac::OPCUA::OPCUA_Conn::NewData(bool, unsigned char const
   *, unsigned char const*) (opcua_pac.cc:48)
==14061==    by 0xBAFBDC: analyzer::opcua::OPCUA_Analyzer::DeliverStream(int, unsigned
   char const*, bool) (OPCUA.cc:60)
==14061==    by 0xD4D0D8: analyzer::Analyzer::NextStream(int, unsigned char const*,
   bool) (Analyzer.cc:245)
==14061==    by 0xD4D542: analyzer::Analyzer::ForwardStream(int, unsigned char const*,
   bool) (Analyzer.cc:331)
==14061==    by 0xCA0237: analyzer::tcp::TCP_Reassembler::Deliver(unsigned long, int,
   unsigned char const*) (TCP_Reassembler.cc:455)
==14061==  Address 0xe811168 is 0 bytes after a block of size 552 alloc'd
==14061==    at 0x4C2E216: operator new(unsigned long) (vg_replace_malloc.c:334)
==14061==    by 0xBC98D9: __gnu_cxx::new_allocator<binpac::OPCUA::READ*>::allocate(
   unsigned long, void const*) (new_allocator.h:104)
==14061==    by 0xBC932B: std::allocator_traits<std::allocator<binpac::OPCUA::READ*>
   >::allocate(std::allocator<binpac::OPCUA::READ*>&, unsigned long) (alloc_traits.h
   :491)
==14061==    by 0xBC8363: std::_Vector_base<binpac::OPCUA::READ*, std::allocator<binpac
   ::OPCUA::READ*> >::_M_allocate(unsigned long) (stl_vector.h:170)
==14061==    by 0xBC6407: binpac::OPCUA::READ** std::vector<binpac::OPCUA::READ*, std::
   allocator<binpac::OPCUA::READ*> >::_M_allocate_and_copy<std::move_iterator<binpac::
   OPCUA::READ**> >(unsigned long, std::move_iterator<binpac::OPCUA::READ**>, std::
   move_iterator<binpac::OPCUA::READ**>) (stl_vector.h:1224)
==14061==    by 0xBC44AC: std::vector<binpac::OPCUA::READ*, std::allocator<binpac::
   OPCUA::READ*> >::reserve(unsigned long) (vector.tcc:75)
==14061==    by 0xBB8A33: binpac::OPCUA::ARRAY_OF_READ::Parse(unsigned char const*,
   unsigned char const*, int) (opcua_pac.cc:2719)
==14061==    by 0xBB8732: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
   *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2652)
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
   char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
   unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
   binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
   unsigned char const*) (opcua_pac.cc:4726)
==14061==
==14061== Invalid read of size 8
==14061==    at 0xBC2220: binpac::OPCUA::read_vector(binpac::OPCUA::ARRAY_OF_READ*) (
   opcua_pac.cc:5210)
==14061==    by 0xBBEFA0: binpac::OPCUA::OPCUA_Flow::proc_opcua_read_request(binpac::
   OPCUA::OPCUA_READ_REQUEST*) (opcua_pac.cc:4867)
```

```
==14061==    by 0xBB879B: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
    *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2660)
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
    char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
    unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
    binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
    unsigned char const*) (opcua_pac.cc:4726)
==14061==    by 0xBB1FB7: binpac::OPCUA::OPCUA_Conn::NewData(bool, unsigned char const
    *, unsigned char const*) (opcua_pac.cc:48)
==14061==    by 0xBAFBDC: analyzer::opcua::OPCUA_Analyzer::DeliverStream(int, unsigned
    char const*, bool) (OPCUA.cc:60)
==14061==    by 0xD4D0D8: analyzer::Analyzer::NextStream(int, unsigned char const*,
    bool) (Analyzer.cc:245)
==14061==    by 0xD4D542: analyzer::Analyzer::ForwardStream(int, unsigned char const*,
    bool) (Analyzer.cc:331)
==14061==    by 0xCA0237: analyzer::tcp::TCP_Reassembler::Deliver(unsigned long, int,
    unsigned char const*) (TCP_Reassembler.cc:455)
==14061==  Address 0xe811188 is 24 bytes after a block of size 560 in arena "client"
==14061==
==14061== Invalid read of size 8
==14061==    at 0xBC32F4: binpac::OPCUA::READ::node_id() const (opcua_pac.h:1069)
==14061==    by 0xBC0C0C: binpac::OPCUA::read(binpac::OPCUA::READ*) (opcua_pac.cc:5068)
==14061==    by 0xBC222A: binpac::OPCUA::read_vector(binpac::OPCUA::ARRAY_OF_READ*) (
    opcua_pac.cc:5210)
==14061==    by 0xBBEFA0: binpac::OPCUA::OPCUA_Flow::proc_opcua_read_request(binpac::
    OPCUA::OPCUA_READ_REQUEST*) (opcua_pac.cc:4867)
==14061==    by 0xBB879B: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
    *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2660)
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
    char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
    unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
    binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
    unsigned char const*) (opcua_pac.cc:4726)
==14061==    by 0xBB1FB7: binpac::OPCUA::OPCUA_Conn::NewData(bool, unsigned char const
    *, unsigned char const*) (opcua_pac.cc:48)
==14061==    by 0xBAFBDC: analyzer::opcua::OPCUA_Analyzer::DeliverStream(int, unsigned
    char const*, bool) (OPCUA.cc:60)
==14061==    by 0xD4D0D8: analyzer::Analyzer::NextStream(int, unsigned char const*,
    bool) (Analyzer.cc:245)
==14061==  Address 0x270 is not stack'd, malloc'd or (recently) free'd
==14061==
==14061==
==14061== Process terminating with default action of signal 11 (SIGSEGV)
==14061==  Access not within mapped region at address 0x270
==14061==    at 0xBC32F4: binpac::OPCUA::READ::node_id() const (opcua_pac.h:1069)
==14061==    by 0xBC0C0C: binpac::OPCUA::read(binpac::OPCUA::READ*) (opcua_pac.cc:5068)
==14061==    by 0xBC222A: binpac::OPCUA::read_vector(binpac::OPCUA::ARRAY_OF_READ*) (
    opcua_pac.cc:5210)
==14061==    by 0xBBEFA0: binpac::OPCUA::OPCUA_Flow::proc_opcua_read_request(binpac::
    OPCUA::OPCUA_READ_REQUEST*) (opcua_pac.cc:4867)
==14061==    by 0xBB879B: binpac::OPCUA::OPCUA_READ_REQUEST::Parse(unsigned char const
    *, unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:2660)
```

```
==14061==    by 0xBBC6FC: binpac::OPCUA::MESSAGE::Parse(unsigned char const*, unsigned
    char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4034)
==14061==    by 0xBBD138: binpac::OPCUA::OPCUA_MESSAGE::Parse(unsigned char const*,
    unsigned char const*, binpac::OPCUA::ContextOPCUA*, int) (opcua_pac.cc:4287)
==14061==    by 0xBBD9C7: binpac::OPCUA::OPCUA_PDU::ParseBuffer(binpac::FlowBuffer*,
    binpac::OPCUA::ContextOPCUA*) (opcua_pac.cc:4494)
==14061==    by 0xBBE5C4: binpac::OPCUA::OPCUA_Flow::NewData(unsigned char const*,
    unsigned char const*) (opcua_pac.cc:4726)
==14061==    by 0xBB1FB7: binpac::OPCUA::OPCUA_Conn::NewData(bool, unsigned char const
    *, unsigned char const*) (opcua_pac.cc:48)
==14061==    by 0xBAFBDC: analyzer::opcua::OPCUA_Analyzer::DeliverStream(int, unsigned
    char const*, bool) (OPCUA.cc:60)
==14061==    by 0xD4D0D8: analyzer::Analyzer::NextStream(int, unsigned char const*,
    bool) (Analyzer.cc:245)
==14061==  If you believe this happened as a result of a stack
==14061==  overflow in your program's main thread (unlikely but
==14061==  possible), you can try to increase the size of the
==14061==  main thread stack using the --main-stacksize= flag.
==14061==  The main thread stack size used in this run was 8388608.
==14061==
==14061== HEAP SUMMARY:
==14061==     in use at exit: 42,609,541 bytes in 613,525 blocks
==14061==   total heap usage: 1,413,304 allocs, 799,779 frees, 130,921,303 bytes
    allocated
==14061==
==14061== LEAK SUMMARY:
==14061==    definitely lost: 667,968 bytes in 34,806 blocks
==14061==    indirectly lost: 10,401,603 bytes in 144,946 blocks
==14061==      possibly lost: 18,008 bytes in 278 blocks
==14061==    still reachable: 31,521,962 bytes in 433,495 blocks
==14061==                       of which reachable via heuristic:
==14061==                         multipleinheritance: 128 bytes in 2 blocks
==14061==         suppressed: 0 bytes in 0 blocks
==14061== Rerun with --leak-check=full to see details of leaked memory
==14061==
==14061== For counts of detected and suppressed errors, rerun with: -v
==14061== Use --track-origins=yes to see where uninitialised values come from
==14061== ERROR SUMMARY: 72 errors from 4 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

References

[1] binpac pull request. https://github.com/bro/binpac/pull/4.

[2] Binpack quickstart. https://github.com/grigorescu/binpac_quickstart/.

[3] Bro - modbus. https://www.bro.org/sphinx/script-reference/proto-analyzers.html#bro-modbus.

[4] Bro analyzers. https://www.bro.org/sphinx/script-reference/proto-analyzers.html.

[5] Bro cluster architecture. https://www.bro.org/sphinx/cluster/index.html.

[6] Bro community projects. https://www.bro.org/development/projects/.

[7] Bro introduction. https://www.bro.org/sphinx/intro/.

[8] Bro on github. https://github.com/bro/bro.

[9] Bro plug-ins. https://github.com/bro/bro-plugins.

[10] Bro types. https://www.bro.org/sphinx/script-reference/types.html.

[11] Extending bro with builtin functions (bif). https://www.bro.org/development/howtos/bif-doc/.

[12] Field devices. http://www2.emersonprocess.com/en-US/brands/edservices/fielddevices/Pages/fielddevices.aspx.

[13] Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/.

[14] Gnu ddd - graphical front-end for command-line debuggers. https://www.gnu.org/software/ddd/.

[15] Guide to industrial control systems (ics) security. Technical report, National Institute of Standards and Technology. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf.

[16] Heartbleed security advisory. https://www.openssl.org/news/secadv/20140407.txt.

[17] Heartbleed site. http://heartbleed.com/.

[18] Iana service names and port numbers. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml.

[19] Iec 61158-1:2014. https://webstore.iec.ch/publication/4624.

[20] The industrial control systems cyber emergency response team (ics-cert). https://ics-cert.us-cert.gov/.

[21] Internet world statistics. http://www.internetworldstats.com/emarketing.htm.

[22] The invisible neutron threat. https://www.lanl.gov/science/NSS/issue1_2012/story4full.shtml.

[23] List of opc foundation unified architecture open source implementations. https://github.com/open62541/open62541/wiki/List-of-Open-Source-OPC-UA-Implementations.

[24] Official opc ua stack reference implementation. https://github.com/OPCFoundation/UA-AnsiC.

[25] Ofree stuff - opc ua. http://www.opcconnect.com/uafree.php.

[26] Opc - ole for process control. https://wiki.wireshark.org/OPC.

[27] Opc and opc ua explained. https://www.novotek.com/en/solutions/kepware-communication-platform/opc-and-opc-ua-explained.

[28] Opc foundation unified architecture open source implementation. http://open62541.org/.

[29] Opc foundation unified architecture reference implementations. https://github.com/OPCFoundation/UA-.NET.

[30] Opc ua - overview. http://www.rtaautomation.com/technologies/opcua/.

[31] Opc ua connects your systems. https://downloads.prosysopc.com/downloads/automation_xx_seminar_opcua_connects_your

[32] Opc-ua: Industrial interoperability for iot. https://opcfoundation.org/wp-content/uploads/2016/05/OPC-UA-Interoperability-For-Industrie4-and-IoT-EN-v5.pdf.

[33] Opc unified architecture. https://opcfoundation.org/about/opc-technologies/opc-ua/.

[34] Packet acquisition and control (pacf). https://www.bro.org/development/projects/pacf.html.

[35] S7 communication (s7comm). https://wiki.wireshark.org/S7comm.

[36] Simple opc-ua gui client. https://github.com/FreeOpcUa/opcua-client-gui.

[37] Stuxnet, the real story. http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet.

[38] Unified automation - opc ua clients. https://www.unified-automation.com/downloads/opc-ua-clients.html.

[39] Valgrind - instrumentation framework for building dynamic analysis tools. http://valgrind.org/.

[40] Wireshark opc ua bug. https://www.wireshark.org/lists/wireshark-bugs/200909/msg00104.html.

[41] Wireshark opc ua plug-in. http://www.ascolab.com/en/home-news/147.html.

[42] Wireshark security advisories. https://www.wireshark.org/security/.

[43] Security and privacy controls for federal information systems and organizations. Technical report, National Institute of Standards and Technology, 2013. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf.

[44] Opc ua - part 1 overview and concepts. Technical report, OPC UA Foundation, 2017. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/.

[45] Opc ua - part 2 security model. Technical report, OPC UA Foundation, 2017. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-2-security-model/.

[46] Opc ua - part 3: Address space model. Technical report, OPC UA Foundation, 2017. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-3-address-space-model/.

[47] Opc ua - part 4: Services. Technical report, OPC UA Foundation, 2017. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-4-services/.

[48] Opc ua - part 6: Mappings. Technical report, OPC UA Foundation, 2017. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-6-mappings/.

[49] Open platform communications unified architecture security analysis. Technical report, Federal office for information security, 2017. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/OPCUA/OPCUA.html.

[50] Rubina Akter Anik Barua, Mohammed Minhazul Hoque. Embedded systems: Security threats and solutions. *American Journal of Engineering Research*, 2014. http://www.ajer.org/papers/v3(12)/P031201190123.pdf.

[51] IEEE ComSoc Arno Claassen, Sebastian Rohjans member, IEEE ComSoc Sebastian Lehnhoff Member, and PES. Application of the opc ua for the smart grid. *Innovative Smart Grid Technologies*, 2011. http://ieeexplore.ieee.org/document/6162627/?reload=true.

[52] K.N. Levitt B. Mukherjee, L.T. Heberlein. Network intrusion detection. *IEEE Network, note=*, 1994.

[53] Andrew Clark Christopher Kruegel, Richard Lippmann. Recent advances in intrusion detection. *RAID: International Workshop on Recent Advances in Intrusion Detection*, 2007. https://link.springer.com/book/10.1007/978-3-540-74320-0.

[54] Prof. Klaus Bender Daniel Grossmann and Benjamin Danzer. Opc ua based field device integration. *SICE Annual Conference 2008*, 2008. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4654789.

[55] Stefan Mangard Daniel Gruss, Clementine Maurice. Rowhammer.js: A remote software-induced fault attack in javascript, 2016. http://link.springer.com/chapter/10.1007/978-3-319-40667-1_15.

[56] Nayeem Islam Elisa Bertino. Botnets and internet of things security. *Computer*, 2017. http://ieeexplore.ieee.org/abstract/document/7842850/?reload=true.

[57] Julius Pfrommer Markus Graube Leon Urbas Florian Palm, Sten Gruner. Open source as enabler for opc ua in industrial automation, 2015.

[58] Thomas Hadlich. Providing device integration with opc ua. *Industrial Informatics*, 2007. http://ieeexplore.ieee.org/document/4053398/.

[59] Pan Dongbo Huang Renjie, Liu Feng. Research on opc ua security.

[60] Slaven Marusica Marimuthu Palaniswamia Jayavardhana Gubbia, Rajkumar Buyyab. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 2013.

[61] Vern Paxson. Bro: A system for detecting network intruders in real-time). *USENIX*, 1998. https://www.icir.org/vern/papers/bro-CN99.pdf.

[62] Somesh Jha Randy Smith, Cristian Estan. Backtracking algorithmic complexity attacks against a nids. *Computer Security Applications Conference*, 2006. http://ieeexplore.ieee.org/abstract/document/4041157/.

[63] Martin Roesch. Snort - lightweight intrusion detection for networks, 1999. http://static.usenix.org/publications/library/proceedings/lisa99/full_papers/roesch/roesch.pdf.

[64] Robin Sommer Larry Peterson Ruoming Pang, Vern Paxson. binpac: A yacc for writing application protocol parsers, 2006. http://dl.acm.org/citation.cfm?id=1177119.

[65] B.P. Miller S. Rubin, S. Jha. Automatic generation and analysis of nids attacks. *Computer Security Applications Conference*, 2005. http://ieeexplore.ieee.org/document/1377213.

[66] Seppo Kuikka Tom Hannelius, Mikko Salmenpera. Roadmap to adopting opc ua. *Industrial Informatics*, 2008. http://ieeexplore.ieee.org/document/4618203/.